МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

Факультет електроенерготехніки та автоматики

МЕТОДИЧНІ ВКАЗІВКИ ТА ЗАВДАННЯ
до виконання лабораторних робіт з дисципліни

«ОБЧИСЛЮВАЛЬНА ТЕХНІКА ТА ПРОГРАМУВАННЯ. ЧАСТИНА 2»
для студентів спеціальності
141 Електроенергетика, електротехніка та електромеханіка

(навчальне електронне видання)

НТУУ “КПІ”
2016

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

Факультет електроенерготехніки та автоматики

МЕТОДИЧНІ ВКАЗІВКИ ТА ЗАВДАННЯ
до виконання лабораторних робіт з дисципліни

«ОБЧИСЛЮВАЛЬНА ТЕХНІКА ТА ПРОГРАМУВАННЯ. ЧАСТИНА 2»
для студентів спеціальності
141 Електроенергетика, електротехніка та електромеханіка

(навчальне електронне видання)

Рекомендовано Вченою радою
факультету електроенерготехніки та автоматики

НТУУ “КПІ”
2016

Методичні вказівки виконання лабораторних робіт з дисципліни «Обчислювальна техніка та програмування. Частина 2» для студентів спеціальності 141 Електроенергетика, електротехніка та електромеханіка / Уклад.: Д.В. Настенко, А.Б. Нестерко, Г.О. Труніна – Київ: НТУУ "КПІ", 2016.

Гриф надано Вченою радою ФЕА НТУУ "КПІ"
(Протокол № 10 від 30 червня 2016 р.)

Навчальне електронне видання

МЕТОДИЧНІ ВКАЗІВКИ ТА ЗАВДАННЯ

до виконання лабораторних робіт з дисциплін

«ОБЧИСЛЮВАЛЬНА ТЕХНІКА ТА ПРОГРАМУВАННЯ. ЧАСТИНА 2»

для студентів спеціальності
141 Електроенергетика, електротехніка та електромеханіка

Укладачі: Настенко Дмитро Васильович, ст. викл.,
Нестерко Артем Борисович, к.т.н., ст. викл.
Труніна Ганна Олексіївна, асистент.

Відповідальний
Редактор О.С. Яндульський, професор, д.т.н.

Рецензент Т.Л. Кацадзе, канд. техн. наук

За редакцією укладачів

ЗМІСТ

Вступ

Розвиток економіки, промисловості, науки і техніки, сфери освіти сьогодні значною мірою залежить від масового запровадження та використання обчислювальної техніки. Це вимагає підготовки і перепідготовки фахівців з програмування і використання персональних комп'ютерів.

Вибір мови програмування C# пояснюється такими чинниками:

- простотою і природністю основних конструкцій мови, що дозволяє швидко її освоїти і створювати алгоритмічно складні програми;
- можливістю використання розвинених засобів подання структур даних, що забезпечує зручність роботи як з числовою, так і з символьною інформацією;
- відповідністю принципам об'єктно-орієнтованого програмування, що робить програми наочними;
- наявністю бібліотеки процедур і функцій для роботи як з текстовою, так і з графічною інформацією, що дозволяє створювати досить складні програми.

ВИМОГИ БЕЗПЕКИ ПІД ЧАС РОБОТИ В ЛАБОРАТОРІЇ ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

1. Ці правила є обов'язковими для всіх студентів та інших осіб, які працюють в лабораторіях постійно чи тимчасово.
2. Перевірка знань цих правил проводиться:
3. обслуговуючого персоналу – завідуючим лабораторією;
4. студентів – викладачами, які є керівниками лабораторних робіт.
5. Після перевірки знань цих правил, кожен з тих, хто працює в лабораторії ставить свій підпис в спеціальному журналі. Без цього підпису ніхто до роботи в лабораторії не повинен бути допущеним.
6. Дотримання правил з техніки безпеки повинно гуртуватися на високій свідомості всіх, хто працює в лабораторіях. Кожен, хто помітить порушення правил, а також несправність, яка являє собою небезпеку для людей і обладнання, повинен сповістити про це керівника.
7. Робота в лабораторії по виконанню конкретного завдання може проводитися тільки після ретельного ознайомлення студентів з обладнанням і роботою, та чіткого уявлення про те, які елементи установки будуть під напругою та дотик до яких є небезпечними у стані роботи.
8. Забороняється виконання ремонтних робіт на обладнанні, яке знаходиться під напругою.
9. Забороняється наближатися або торкатися до струмоведучих частин, що обертаються, або усувати несправності без відключення установок.
10. Забороняється проводити переключення в схемі, яка знаходиться під напругою.
11. Перевірку наявності напруги дозволяється проводити тільки за допомогою вольтметра.

12. Апарати управління та вимірювальні прилади слід розташовувати так, щоб було зручно вести спостереження за приладами, не перегинаючись через проводи та апарати.
13. У випадку виникнення будь-яких несправностей необхідно негайно вимкнути живленим установки та сповістити керівника занять про це.
14. Кнопки управління, рубильники встановлювати в легкодоступних місцях для швидкого виключення схеми.
15. У випадку припинення подачі електроенергії в лабораторію всі установки в лабораторії обов'язково вимикаються на робочих місцях.
16. В лабораторіях категорично забороняється:
    - Палити в усіх приміщеннях, крім спеціально відведених для цього місць;
    - Прокладати без дозволу постійні та тимчасові лінії;
    - Користуватися побутовими електронагрівальними приладами;
    - Користуватися зіпсованим електрообладнанням, саморобними запобіжниками, провідниками із зіпсованою ізоляцією та саморобними електросвітильниками;
    - Проводити в непристосованих приміщеннях обмивку та фарбування деталей горючими рідинами та фарбниками;
    - Зберігати паливно-мастильні матеріали, хімікати та інші горючі речовини;
    - Загромаджувати проходи в лабораторіях;
17. Всі, хто працює в лабораторії повинні знати, де знаходиться аптечка з медикаментами для наданим першої допомоги.
18. При ураженні людини електричним струмом треба негайно вимкнути напругу, надати першу допомогу та покликати лікаря.

19. Порятунок осіб, які постраждали, залежить від тою, як швидко вони будуть звільнені від електричного струму та як швидко їм буде надано першу допомогу.

20. Першу допомогу необхідно надати негайно на місці події.

21. Переносити людину, яка постраждала в інше місце необхідно тільки в тих випадках, коли небезпека продовжує загрожувати або надання допомоги на місці неможливе.

22. При відсутності у постраждалого дихання, серцебиття, пульсу ніколи не треба ставити під сумнів необхідність першої допомоги, тому що при ураженні електричним струмом смерть часто буває несправжньою. Тільки лікар може дати висновок про смерть постраждалого.

23. До приїзду лікаря постраждалому необхідно надати допомогу і провести штучне дихання з дотриманням всіх правил надання першої допомоги.

24. Про випадок негайно треба сповістити керівництво кафедри, деканату та інституту.

25. Недотримання цих вимог не дозволяється. Якщо розпорядження суперечить діючим правилам, необхідно надати роз'яснення з приводу неухильною виконання цих правил і довести це до відома керівництва.

ЗАГАЛЬНІ ВИМОГИ ДО ВИКОНАННЯ ЛАБОРАТОРНИХ РОБІТ

Підготовка до кожної роботи проводиться в поза аудиторний час. Студенти знайомляться із загальними відомостями, пишуть необхідні програми відповідно до отриманого варіанта індивідуального завдання.

Усі роботи виконуються на мові програмування C#. Під час занять студенти проводять тестування написаних програм, тобто запускають їх в інтегрованому середовищі розробки на персональних комп'ютерах, займаються налагодженням і виконують необхідні розрахунки.

Після виконання роботи студент оформляє звіт, який складається з таких розділів:

1. Назва, тема і мета роботи
2. Індивідуальне завдання
3. Текст програми
4. Результати виконання програми

Шаблон звіту про виконання лабораторної роботи наведено в додатку А. Робота оформлюється в електронному або друкованому вигляді.

Під час захисту роботи необхідно відповісти на контрольні питання і вміти пояснити роботу програми.

ЛАБОРАТОРНА РОБОТА №1.
КЛАСИ

Мета

Засвоїти основні поняття концепції об’єктно-орієнтованого програмування. Вивчити основні принципи реалізації класів в мові програмування C#. Отримати досвід створення простих класів з полями базових типів даних.

Стислі теоретичні відомості

Клас є типом даних, визначеним користувачем. Він повинен представляти собою одну логічну сутність, наприклад, бути моделлю реального об'єкта або процесу. Елементами класу є дані і функції, призначені для їх обробки.

Опис класу містить ключове слово **class**, за яким слідує його ім'я, а далі у фігурних дужках - тіло класу, тобто список його елементів. Крім того, для класу можна задати його базові класи (предки) і ряд необов'язкових атрибутів і специфікаторів, які визначають різні характеристики класу:

```
[атрибути] [специфікатор] class ім'я_класу [:предки]
тіло класу
```

Обов'язковими є тільки ключове слово **class**, а також ім'я та тіло класу. Ім'я класу задається програмістом за загальними правилами C #. Тіло класу - це список описів його елементів, укладений у фігурні дужки. Список може бути порожнім, якщо клас не містить жодного елемента.

Специфікатори визначають властивості класу, а також доступність класу для інших елементів програми. Можливі значення специфікаторів перераховані в Таблиця 1.1. Клас можна описувати безпосередньо всередині простору імен або всередині іншого класу. В останньому випадку клас називається вкладеним. Залежно від місця опису класу деякі з цих специфікаторів можуть бути заборонені.

Таблиця 1.1 Специфікатори класів

№	Специфікатор	Опис
1	new	Використовується для вкладених класів. Задає новий опис класу, натомість успадкованого від предка. Застосовується в ієрархіях об'єктів
2	public	Доступ не обмежений
3	protected	Використовується для вкладених класів. Доступ лише з елементів даного і похідних класів
4	internal	Доступ лише з даної програми
5	protected internal	Доступ лише з даного і похідних класів або з даної програми
6	private	Використовується для вкладених класів. Доступ лише з елементів класу, всередині якого описаний даний клас
7	abstract	Абстрактний клас. Застосовується в ієрархіях об'єктів
8	sealed	Закритий клас. Застосовується в ієрархіях об'єктів
9	static	Статичний клас

Специфікатори 2-6 називаються специфікаторами доступу. Вони визначають, звідки можна безпосередньо звертатися до даного класу.

У цій роботі вивчаються класи, які описуються в просторі імен безпосередньо (тобто не вкладені класи). Для таких класів допускаються тільки два специфікатора: **public** і **internal**. За замовчуванням, тобто якщо жоден специфікатор доступу не вказаний, мається на увазі специфікатор **internal**.

Клас є узагальненим поняттям, що визначає характеристики і поведінку всіх конкретних об'єктів цього класу, які називають екземплярами, або об'єктами, класу.

Об'єкти створюються явним чи неявним чином, тобто або програмістом, або системою. Програміст створює екземпляр класу за допомогою операції **new**, наприклад:

```
Book a = new Book();
var b = new Book();
```

Для кожного об'єкта при його створенні в пам'яті виділяється окрема область, в якій зберігаються його дані. Крім того, в класі можуть бути присутніми статичні елементи, які існують в єдиному екземплярі для всіх об'єктів класу. Часто статичні дані називають даними класу, а решта - даними екземпляра.

Функціональні елементи класу не тиражуються, тобто завжди зберігаються в єдиному екземплярі. Для роботи з даними класу використовуються методи класу (статичні методи), для роботи з даними примірника – методи примірника, або просто методи.

До цього часу ми використовували в програмах тільки один вид функціональних елементів класу – методи. Поля і методи є основними елементами класу. Крім того, в класі можна задавати цілу гаму інших елементів: властивості, події, індексатори, операції, конструктори, деструктори, а також типи.

Дані, що містяться в класі, можуть бути змінними або константами. Змінні, описані в класі, називаються полями класу.

При описі елементів класу можна також вказувати атрибути і специфікатори задають різні характеристики елементів. Синтаксис опису елемента даних наведений нижче:

```
[атрибути] [специфікатор] [const] тип ім'я
[= початкове значення];
```

Можливі специфікатори полів і констант перераховані в Таблиця 1.2. Для констант можна використовувати тільки специфікатори 1-6.

За замовчуванням елементи класу вважаються закритими (**private**). Для полів класу цей вид доступу є кращим, оскільки поля визначають внутрішню

будову класу, яке має бути прихована від користувача. Всі методи класу мають безпосередній доступ до його закритих полів.

Таблиця 1.2 Специфікатори полів і констант

№	Специфікатор	Опис
1	new	Нове опис поля, яке приховує успадкований елемент класу
2	public	Доступ до елементу не обмежений
3	protected	Доступ лише з даного і похідних класів
4	internal	Доступ лише з даної збірки
5	protected internal	Доступ лише з даного і похідних класів і з даної збірки
6	private	Доступ лише з даного класу
7	static	Одне поле для всіх екземплярів класу
8	readonly	Поле доступне тільки для читання
9	volatile	Поле може змінюватися іншим процесом або системою

Поля, описані зі специфікатором **static**, а також константи існують в єдиному екземплярі для всіх об'єктів класу, тому до них звертаються не через ім'я екземпляра, а через ім'я класу. Якщо клас містить тільки статичні елементи, екземпляр класу створювати не потрібно.

Звернення до полю класу виконується за допомогою операції доступу (точка). Праворуч від точки задається ім'я поля, ліворуч - ім'я екземпляра для звичайних полів або ім'я класу для статичних.

Всі поля спочатку автоматично ініціалізуються нулем відповідного типу (наприклад, полям типу **int** присвоюється 0, а посиланням на об'єкти - значення **null**). Після цього полю присвоюється значення, задане при його явній ініціалізації. Завдання початкових значень для статичних полів виконується при ініціалізації класу, а звичайних – при створенні екземпляру.

Поля зі специфікатором **readonly** призначені тільки для читання. Встановити значення такого поля можна або при його описі, або в конструкторі.

Робоче завдання

Навчитися створювати прості класи з полями даних базових типів.

Хід роботи

Написати програму на мові C#, в якій описано клас, що відповідає об’єкту відповідно до варіанту індивідуального завдання. Створити три екземпляри описаного класу з різними значеннями полів та вивести ці значення на екран.

Індивідуальне завдання

1. Створити клас, що містить відомості про студента: прізвище, ім’я, по батькові, рік народження, зріст.
2. Створити клас, що містить відомості про області України: назва, площа, населення, адміністративний центр, населення в адміністративному центрі.
3. Створити клас, що містить відомості про модель телефону: виробник, діагональ екрана, кількість процесорних ядер, ціна.
4. Створити клас, що містить відомості про електродвигун: тип, номінальна потужність, кількість фаз, маса.
5. Створити клас, що містить відомості про монітор комп’ютера: модель, розширення по горизонталі, розширення по вертикалі, яскравість, ціна.
6. Створити клас, що містить відомості про лінію електропередачі: тип кабелю, довжина, активний опір, реактивний опір, кількість фаз.
7. Створити клас, що містить відомості про будинок: адреса, кількість поверхів, загальна площа, кількість мешканців.
8. Створити клас, що містить відомості про футбольну команду: назва, місто, країна, кількість набраних очок за сезон, середня зарплата футболістів.

9. Створити клас, що містить відомості про літак: модель, максимальна швидкість, максимальна висота польоту, кількість пасажирів.
10. Створити клас, що містить відомості про навчальний предмет: назва, кількість семестрів, кількість лекцій, прізвище викладача.
11. Створити клас, що містить відомості про планету сонячної системи: назва, маса, діаметр, кількість супутників.
12. Створити клас, що містить відомості про провід ліній електропередачі: марка, площа перерізу, питомий активний опір, матеріал.
13. Створити клас, що містить відомості про квартиру: номер, площа, кількість кімнат, споживання електроенергії за місяць.
14. Створити клас, що містить відомості про модель ноутбука: назва, виробник, рік випуску, час автономної роботи.
15. Створити клас, що містить відомості про студента: прізвище, номер телефону та екзаменаційні оцінки з трьох предметів.
16. Створити клас, що містить відомості про прізвища, табельний № та зарплату працівника заводу.
17. Створити клас, що містить відомості про автора, назву книги, кількість сторінок та рік її видання.
18. Створити клас, що містить коефіцієнти квадратного рівняння, та його запис у текстовому вигляді.
19. Створити клас, що містить відомості про розклад руху потягів: номер потяга, назва станції, час прибуття, час відправлення.
20. Створити клас, що містить відомості про здачу студентом екзамену: номер групи, прізвище студента, предмет, оцінка.
21. Створити клас, що містить відомості про запис студента в бібліотеку: прізвище, ім’я, рік народження, дата запису до бібліотеки.

22. Створити клас, що містить відомості про номер автомобіля, марку автомобіля, рік випуску та колір.
23. Створити клас, що містить відомості про назву країни, площу країни, населення країни, назву столиці, населення столиці.
24. Створити клас, що містить відомості про розклад занять: назва предмета, № пари, день тижня, прізвище викладача.
25. Створити клас, що містить відомості про трансформатор: тип, номінальна потужність, напруга первинної обмотки, напруга вторинної обмотки, кількість фаз.

Контрольні питання

1. Що таке клас?
2. Що таке поле класу?
3. Що таке екземпляр класу?
4. Наведіть загальну форму опису класу.
5. Наведіть загальну форму опису елемента даних класу.
6. Що таке тіло класу?
7. Чи можна вкладати клас у інший клас?
8. Що таке оператор доступу до елементу класу?
9. Що таке специфікатори доступу? Наведіть приклади.

ЛАБОРАТОРНА РОБОТА №2.
РОБОТА З ПОЛЯМИ КЛАСУ. СПЕЦИФІКАТОРИ ДОСТУПУ

Мета

Засвоїти основні поняття організації управління доступом до даних класу. Вивчити основні принципи роботи специфікаторів доступу в мові програмування C#. Отримати досвід створення методів для організації роботи з закритими полями класу.

Стислі теоретичні відомості

У мові C #, по суті, є два типи елементів класу: відкриті і закриті (хоча насправді існує ще декілька інших різновидів). Доступ до відкритого члену можна здійснювати з коду, визначеного за межами класу. Саме цей тип – **public** – елемента класу використовувався в попередній лабораторній роботі. Закритий елемент класу доступний тільки методами, визначеними в самому класі. За допомогою закритих елементів і організовується управління доступом до даних класу.

Обмеження доступу до елементів класу є одним із основних принципів об'єктно-орієнтованого програмування, оскільки дозволяє виключити некоректне використання об'єкта. Виконуючи доступ до закритих даних тільки за допомогою чітко визначених методів, можна виключити присвоювання невірних значень цих даних, забезпечуючи, наприклад, перевірку діапазону представлення чисел. Тоді правильно реалізований клас утворює свого роду "чорний ящик", яким можна користуватися без втручання у внутрішній механізм його роботи.

Як уже було зазначено, управління доступом в мові C# організовується за допомогою таких специфікаторів доступу, як **public**, **private**, **protected** і **internal**. Коли елемент класу позначається специфікатором **public**, він стає доступним з будь-якого іншого коду в програмі, включаючи і методи, визначені в інших класах. Коли ж елемент класу позначається специфікатором

private, він може бути доступний тільки іншим членам цього класу. Отже, методи з інших класів не мають доступу до закритого елементу (**private**) даного класу.

Для того щоб стали більш зрозумілими відмінності між специфікаторами **public** і **private**, розглянемо наступний приклад програми:

```
using System;

namespace ClassElementTest
{
    class Lamp
    {
        private double power; // Закритий доступ, вказується явно
        double current; // Закритий доступ за замовчуванням
        public double NominalPower; // Відкритий доступ

        // Методи, яким доступні поля даного класу
        // Метод класу може мати доступ до закритого члену цього ж класу
        public void SetPower(double value)
        {
            if (value >= 0 && value <= NominalPower)
            {
                power = value;
                current = value / 220; // Приймемо, що I=P/U, U=220 В
            }
        }
        public double GetPower()
        {
            return power;
        }
        public double GetCurrent()
        {
            return current;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            var myLamp=new Lamp();
            // Поле NominalPower класу Lamp доступне безпосередньо,
            // оскільки воно є відкритим
            myLamp.NominalPower = 100;
            // Доступ до полів power і current класу Lamp дозволений
            // тільки за допомогою його методів
            myLamp.SetPower(85.3);
            Console.WriteLine("Потужність лампи: {0} Вт, струм лампи: {1} А",
                myLamp.GetPower(), myLamp.GetCurrent());
            // Наступні види доступу до полів power і current класу Lamp не дозволяються
            // myLamp.power = -10;  // Помилка!  power - закритий член!
            // Console.WriteLine(myLamp.current);  // Помилка!  current - закритий член!
        }
    }
```

```
}
```

Як бачите, в класі *Lamp* поле *power* вказано явно як **private**, поле *current* стає **private** за замовчуванням, а поле *NominalPower* вказано як **public**. Таким чином, поля *power* і *current* недоступні безпосередньо з коду за межами даного класу, оскільки вони є закритими. Зокрема, ними не можна користуватися безпосередньо в класі *Program*. Вони доступні тільки за допомогою таких відкритих (**public**) методів, як *SetPower* () і *GetPower* (). Так, якщо видалити символи коментаря на початку рядка коду:

```
        // myLamp.power = -10;  // Помилка!  power - закритий член!
```

то наведена вище програма не скомпілюється через порушення правил доступу. Але незважаючи на те, що член *power* недоступний безпосередньо за межами класу *Lamp*, вільний доступ до нього організовується за допомогою методів, визначених в класі *Lamp*, як наочно показують методи *SetPower* () і *GetPower* (). Таким чином виключається задання некоректних значень полів (у даному випадку від'ємної потужності та потужності, яка більша за номінальну).

Робоче завдання

Навчитися управляти доступом до полів класу та створювати методи для організації роботи з закритими полями класу.

Хід роботи

1. Модифікувати та доповнити програму створену під час виконання лабораторної роботи №1:
    - надати специфікатор доступу **private** двом, вибраним на ваш розсуд, полям даних класу;
    - створити відповідні методи для задання та зчитування значень закритих полів;
    - доповнити клас методом *ToString* () що забезпечує отримання інформації про клас у текстовому вигляді та дає змогу

використовувати спрощений запис методів *WriteLine* () для екземплярів класу. Приклад методу:

```
public override string ToString()
{
    return string.Format("Потужність лампи: {0} Вт, струм лампи: {1} А",
        power, current);
}
```

Результати використання:

```
// Старий запис методу WriteLine ()
Console.WriteLine("Потужність лампи: {0} Вт, струм лампи: {1} А",
    myLamp.GetPower(), myLamp.GetCurrent());
// Новий запис методу WriteLine ()
Console.WriteLine(myLamp);
```

- модифікувати відповідним чином метод *Main* () класу *Program*.

Індивідуальне завдання

Відповідно до завдання лабораторної роботи №1.

Контрольні питання

1. Що таке закритий елемент класу?
2. Що таке відкритий елемент класу?
3. Що таке специфікатор доступу?
4. Яка різниця між специфікаторами доступу **public** та **private**?
5. Чому для полів класу доцільно використовувати специфікатор доступу **private?**
6. Як забезпечити доступ до закритих полів класу з методів інших класів?
7. Чи можна для поля класу задати одночасно два специфікатори доступу?

ЛАБОРАТОРНА РОБОТА №3.
КОНСТРУКТОРИ КЛАСІВ

Мета

Вивчити основні принципи роботи конструкторів класів в мові програмування C#. Отримати досвід створення конструкторів для ініціалізації екземплярів класу.

Стислі теоретичні відомості

У наведених вище прикладах програм змінні екземпляра кожного об'єкта типу доводилося задавати вручну, використовуючи, зокрема, наступну послідовність операторів:

```
house.Occupants = 4;
house.Area = 2500;
house.Floors = 2;
```

Такий прийом зазвичай не застосовується в професійно написаному коді C#. Крім того, він супроводжується помилками, коли можна просто забути ініціалізувати одне з полів. Існує кращий спосіб вирішити подібну задачу: скористатися конструктором.

Конструктори призначені для ініціалізації об'єктів. Вони викликаються автоматично при створенні об'єкта класу за допомогою операції **new**. Ім'я конструктора обов’язково збігається з ім'ям класу.

Нижче наведена загальна форма опису конструктора:

```
[специфікатори] ім'я_класу ([список_параметрів])
тіло_конструктора
```

Властивості конструкторів:

- конструктор не повертає значення, навіть типу **void**;
- клас може мати кілька конструкторів з різними параметрами для різних видів ініціалізації;
- якщо програміст не вказав жодного конструктора, то застосовується конструктор за замовчуванням, у якому полям

значущих типів присвоюється нуль, а полям посилальних типів - значення **null**.

Як правило, конструктор використовується для завдання початкових значень змінних екземпляра, визначених в класі, або ж для виконання будь-яких інших процедур, які потрібні для створення повністю сформованого коректного об'єкта. Для конструктора зазвичай задають специфікатор доступу **public**, оскільки конструктори, в основному, викликаються з інших класів. Список параметрів конструктора може бути порожнім, або містити один чи більше параметрів.

У всіх класів є конструктори, незалежно від того, визначаються вони явно чи ні. В C# автоматично надається конструктор, який використовується за замовчуванням і у якому всі змінні екземпляра ініціалізуються їх значеннями за замовчуванням. Для більшості типів даних значенням за замовчуванням є нульовим, для типу **bool** – значення **false**, а для типів-посилань – значення **null**. Як тільки ви визначите свій власний конструктор, то конструктор за замовчуванням більше не викликається.

Часто буває зручно задати в класі кілька конструкторів, щоб забезпечити можливість ініціалізації об'єктів різними способами. У цьому випадку використовуються параметризовані конструктори. У конструкторів параметри задаються таким же чином, як і в методах. Для цього достатньо оголосити їх в дужках після імені конструктора. Всі конструктори класу повинні мати різні сигнатури.

Після знайомства з конструкторами класів, повернемося до оператора **new**. Для класів загальна форма оператора **new** така:

```
new ім'я_класу ([список_параметрів])
```

де ім'я_класу позначає ім'я класу, що реалізується у вигляді його екземпляра. Якщо в класі не визначений його власний конструктор, то в операторі **new** буде використаний конструктор, що надається в C# за замовчуванням.

Робоче завдання

Навчитися створювати конструктори класів в мові програмування C# та отримати досвід використання конструкторів для ініціалізації екземплярів класу.

Хід роботи

1. Модифікувати та доповнити програму створену під час виконання лабораторної роботи №2:
    - надати специфікатор доступу **private** всім іншим полям даних класу;
    - для описаного раніше класу створити конструктор без параметрів для ініціалізації полів певними початковими значеннями (значення вибрати самостійно);
    - додатково створити два параметризованих конструктори з різними наборами параметрів для ініціалізації відповідних полів початковими значеннями;
    - модифікувати метод *Main* () класу *Program* шляхом заміни операторів присвоювання значень полів на виклики відповідних конструкторів.

Індивідуальне завдання

Відповідно до завдання лабораторної роботи №1.

Контрольні питання

1. Що таке конструктор класу?
2. Що таке параметризований конструктор?
3. Які специфікатори доступу можливо використовувати для конструкторів?

4. Чому для конструкторів класу доцільно використовувати специфікатор доступу **public?**
5. Як відбувається ініціалізація полів класу за допомогою конструкторів?
6. Як відбувається виклик конструктора класу?
7. Чи може клас містити більше одного конструктора?
8. Що таке конструктор за замовчуванням? Як він працює?
9. Наведіть загальний синтаксис опису конструктора класу.

ЛАБОРАТОРНА РОБОТА №4.
ВЛАСТИВОСТІ

Мета

Вивчити основні принципи роботи властивостей, як елементів класів в мові програмування C#. Отримати досвід створення властивостей для доступу до даних класу.

Стислі теоретичні відомості

Властивість є елементом класу, що поєднує в собі поле з методами доступу до нього. Властивості служать для організації доступу до полів класу. Як правило, властивість пов'язана із закритим полем класу і визначає методи його отримання та установки.

Властивість складається з імені і так званих аксесорів **get** і **set**. Аксесори служать для зчитування і установки значення змінної. Головна перевага властивості над спеціальними методами полягає в тому, що її ім'я може бути використано в виразах і операторах присвоювання аналогічно імені звичайної змінної. При зверненні до властивості по імені автоматично викликаються її аксесори **get** і **set**.

Нижче наведена загальна форма властивості:

```
[ атрибути ] [ специфікатори ] тип ім’я_властивості
{
[ специфікатор ] [ get код_доступу ]
[ специфікатор ] [ set код_доступу ]
}
```

де тип позначає конкретний тип властивості, наприклад **int**.

Значення специфікаторів для властивостей і методів аналогічні. Найчастіше властивості оголошуються як відкриті (зі специфікатором **public**), оскільки вони реалізують взаємодію зовнішнього коду з даними класу.

Код доступу є блоком операторів, які виконуються при отриманні (**get**) або установці (**set**) властивості.

Як тільки властивість визначено, будь-яке звернення до властивості по імені приведе до автоматичного виклику відповідного аксесора. Крім того, аксесор **set** має неявний параметр **value**, який містить значення, що присвоюється властивості.

Властивості не визначають місце в пам'яті для зберігання полів, а лише управляють доступом до полів. Сама властивість не описує поле, і тому поле повинно бути визначено незалежно від властивості (винятком з цього правила є автоматичні властивості).

Нижче наведено простий приклад програми, в якій визначається властивість *Power*, призначена для доступу до поля *_power*. В даному прикладі властивість використовується для обмеження діапазону допустимих значень потужності лампи.

```
using System;

namespace ClassElementTest
{
    class Lamp
    {
        private double _power;
        public double NominalPower;

        // Властивість для доступу до поля _power
        public double Power
        {
            get { return _power; }
            set {
                if (value>=0 && value<=NominalPower)
                {
                    _power = value;
                }
            }
        }

        // Властивість без базового поля. Призначена для розрахунку
значення струму
        public double Current
        {
```

```
            get { return _power/220; } // Приймемо, що I=P/U, U=220 В
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            var myLamp = new Lamp();
            myLamp.NominalPower = 100;
            // Задаєм значення за допомогою властивості
            myLamp.Power = 83.5;
            Console.WriteLine("Потужність лампи: {0} Вт, струм лампи:
{1:n3} А",
                myLamp.Power, myLamp.Current);
        }
    }
}
```

Розглянемо наведений вище код більш докладно. У цьому коді визначається закрите поле *_power* і властивість *Power*, котра управляє доступом до поля *_power*.

Властивість *Power* вказано як **public**, а отже, вона доступна з коду за межами класу. Аксесор **get** цієї властивості просто повертає значення поля *_power*, тоді як аксесор **set** встановлює значення в поле *_power* тільки в тому випадку, якщо це значення відповідає обмеженням. Таким чином, властивість *Power* контролює значення, які можуть зберігатися в полі *_power*. У цьому, власне, і полягає основне призначення властивостей.

Починаючи з версії C# 3.0, з'явилася можливість створювати прості властивості, не вдаючись до явного визначення поля, яким управляє властивість. У цьому випадку базове поле для властивості автоматично надає компілятор. Така властивість називається автоматичною і має таку загальну форму:

```
[ атрибути ] [ специфікатори ] тип ім’я_властивості
{get; set; }
```

Зверніть увагу на те, що після позначень аксесорів **get** і **set** відразу ж ставиться крапка з комою, а тіло у них відсутнє. Такий синтаксис вказує

компілятору створити автоматично поле для зберігання значення властивості. Таке поле недоступне безпосередньо і не має імені.

Нижче наведено приклад опису автоматичної властивості:

```
public double Voltage { get; set; }
```

Як бачите, в цьому рядку змінна явно не оголошується. Компілятор автоматично створює анонімне поле, в якому зберігається значення властивості.

На відміну від звичайних властивостей, автоматична властивість не може бути доступною тільки для читання або тільки для запису. При оголошенні цієї властивості необхідно вказувати обидва аксесори – **get** і **set**. Зробити автоматичну властивість доступною тільки для читання або тільки для запису можна оголосивши непотрібний аксесор як **private**.

Незважаючи на очевидні зручності автоматичних властивостей, їх застосування обмежується в основному тими ситуаціями, в яких не потрібно керувати установкою або отриманням значень з полів. Базове поле автоматичної властивості недоступне безпосередньо. Це означає, що на значення, яке може мати автоматична властивість, не можна накласти ніяких обмежень.

Властивостями притаманний ряд обмежень. По-перше, властивість не визначає місце для зберігання даних, і тому не може бути передана методу в якості параметра **ref** або **out**. По-друге, властивість не повинна змінювати стан базової змінної при виклику аксесора **get**. І хоча це правило не перевіряється компілятором, його порушення вважається семантичної помилкою.

За замовчуванням доступність аксесорів **set** і **get** є такою ж, як і у властивості, частиною якої вони є. Так, якщо властивість оголошується як **public**, то за замовчуванням її аксесори **set** і **get** також стають відкритими (**public**). Проте для аксесорів **set** або **get** можна вказати власний специфікатор

доступу, наприклад **private**. Доступність аксесорів, що визначається таким специфікатором, повинна бути більш обмеженою, ніж доступність властивості.

Існує ряд причин, за яких потрібно обмежити доступність аксесорів. Припустимо, що потрібно надати вільний доступ до значення властивості, але разом з тим дати можливість змінювати цю властивість тільки елементам її класу. Для цього достатньо оголосити аксесор **set** даної властивості як **private**.

Робоче завдання

Навчитися створювати властивості в мові програмування C# та отримати досвід використання властивостей для доступу до даних екземплярів класу.

Хід роботи

1. Модифікувати та доповнити програму створену під час виконання лабораторної роботи №3:
    - для двох полів описаного раніше класу створити властивості, що забезпечують можливість їх зчитування та зміни;
    - додатково створити дві автоматичні властивості, які описують додаткові характеристики об’єкта на ваш вибір;
    - створити властивість, доступну тільки для читання, яка розраховує певне значення характеристики об’єкта на ваш вибір;
    - модифікувати конструктори класів шляхом додавання нових властивостей та заміни операторів присвоювання значень полів на оператори присвоювання значень властивостей;
    - доповнити метод *Main* () класу *Program* операторами присвоювання значень властивостям.

Індивідуальне завдання

Відповідно до завдання лабораторної роботи №1.

Контрольні питання

1. Що таке властивість класу?
2. Що таке автоматична властивість?
3. Для чого призначені аксесори **set** і **get**?
4. Які специфікатори доступу можливо використовувати для властивостей?
5. Чи можна використовувати різні специфікатори доступу для аксесорів **set** і **get?**
6. Як відбувається задання та зчитування полів класу за допомогою властивостей?
7. Чи може клас містити властивості, які не пов’язані з окремими полями?
8. Наведіть загальний синтаксис опису властивості.

ЛАБОРАТОРНА РОБОТА №5.
КОЛЕКЦІЇ. КЛАС LIST<T>

Мета

Вивчити основні принципи роботи колекцій в мові програмування C#. Отримати досвід створення узагальнених колекцій для зберігання та роботи з екземплярами класу.

Стислі теоретичні відомості

У бібліотеках більшості сучасних об'єктно-орієнтованих мов програмування представлені стандартні класи, що реалізують основні абстрактні структури даних. Такі класи називаються колекціями, або контейнерами. Для кожного типу колекції визначені методи роботи з її елементами, які не залежать від конкретного типу даних, які зберігаються в колекції, тому один і той же вид колекції можна використовувати для зберігання даних різних типів. Використання колекцій дозволяє скоротити терміни розробки програм і підвищити їх надійність.

Кожен вид колекції підтримує свій набір операцій над даними, і швидкодія цих операцій може бути різним. Вибір виду колекції залежить від того, що потрібно робити з даними в програмі і які вимоги пред'являються до її швидкодії. Наприклад, при необхідності часто вставляти і видаляти елементи з середини послідовності слід використовувати список, а не масив, а якщо включення елементів виконується головним чином в кінець або початок послідовності - черга. Тому вивчення можливостей стандартних колекцій і їх грамотне застосування є необхідними умовами створення ефективних і професійних програм.

У бібліотеці .NET визначені стандартні класи, що реалізують більшість перерахованих раніше абстрактних структур даних. Основні простори імен, в яких описані ці класи, – **System.Collections, System.Collections.Specialized** і **System.Collections.Generic**.

Всі колекції розроблені на основі набору чітко визначених інтерфейсів. Деякі вбудовані реалізації таких інтерфейсів, в тому числі **ArrayList, Hashtable, Stack** і **Queue**, можуть застосовуватися в початковому вигляді і без будь-яких змін. Є також можливість реалізувати власну колекцію, хоча потреба в цьому виникає вкрай рідко.

У середовищі .NET Framework підтримуються п'ять типів колекцій:

- неузагальнені,
- спеціальні,
- з поразрядною організацією,
- узагальнені,
- паралельні.

Неузагальнені колекції реалізують ряд основних структур даних, включаючи динамічний масив, стек, чергу, а також словники, в яких можна зберігати пари "ключ-значення". Відносно неузагальнених колекцій важливо мати на увазі наступне: вони оперують даними типу **object**.

Таким чином, неузагальнені колекції можуть служити для зберігання даних будь-якого типу, причому в одній колекції допускається наявність різнотипних даних. Очевидно, що такі колекції не типізовані, оскільки в них зберігаються посилання на дані типу **object**. Класи і інтерфейси неузагальнених колекцій знаходяться в просторі імен **System.Collections**.

Спеціальні колекції оперують даними конкретного типу або ж роблять це якимось особливим чином. Наприклад, є спеціальні колекції для символьних рядків, а також спеціальні колекції, в яких використовується односпрямований список. Спеціальні колекції оголошуються в просторі імен **System.Collections.Specialized**.

У прикладному інтерфейсі **Collections API** визначена одна колекція з поразрядною організацією - це **BitArray**. Колекція типу **BitArray** підтримує порозрядні операції, тобто операції над окремими двійковими розрядами, наприклад, І йди виключає АБО, а отже, вона істотно відрізняється своїми

можливостями від інших типів колекцій. Колекція типу **BitArray** оголошується в просторі імен **System.Collections**.

Узагальнені колекції забезпечують узагальнену реалізацію декількох стандартних структур даних, включаючи списки, стеки, черги і словники. Такі колекції є типізованими в силу їх узагальненого характеру. Це означає, що в узагальненій колекції можуть зберігатися тільки такі елементи даних, які сумісні за типом з даної колекцією. Завдяки цьому виключається випадкова розбіжність типів. Узагальнені колекції оголошуються в просторі імен **System.Collections.Generic**.

Як правило, класи узагальнених колекцій є узагальненими еквівалентами класів неузагальнених колекцій, хоча це відповідність не є взаємно однозначною. Наприклад, в класі узагальненої колекції **LinkedList** реалізується двонаправлений список, тоді як в неузагальнених еквіваленті його не існує. У деяких випадках одні й ті ж функції існують паралельно в класах узагальнених і неузагальнених колекцій, хоча і під різними іменами. Так, узагальнений варіант класу **ArrayList** називається **List**, а узагальнений варіант класу **HashTable** – **Dictionary**.

У табл. 5.1 перераховані основні колекції, описані в просторі **System.Collections.Generic**.

Таблиця 5.1 – Колекції простору імен System.Collections.Generic

Клас	Опис
`Dictionary <Tkey, TValue>`	Зберігає пари "ключ-значення". Забезпечує такі ж функціональні можливості, як і неузагальнений клас Hashtable
`HashSet <T>`	Зберігає ряд унікальних значень, використовуючи хеш таблицю
`LinkedList <T>`	Зберігає елементи в двонаправленому списку
`List <T>`	Створює динамічний масив. Забезпечує такі ж функціональні можливості, як і неузагальнений клас ArrayList
`Queue <T>`	Створює чергу. Забезпечує такі ж функціональні можливості, як і неузагальнений клас Queue

SortedDictionary <TKey, TValue>	Створює відсортований список з пар "ключ-значення"
SortedList <TKey, TValue>	Створює відсортований список з пар "ключ-значення". Забезпечує такі ж функціональні можливості, як і неузагальнений клас SortedList
SortedSet <T>	Створює відсортовану множину
Stack <T>	Створює стек. Забезпечує такі ж функціональні можливості, як і неузагальнених клас Stack

У колекції можна зберігати не тільки об'єкти вбудованих типів. В них допускається зберігати об'єкти будь-якого типу, включаючи об'єкти класів, що визначаються користувачем.

Завдяки тому, що всі типи успадковують від класу **object**, в неузагальнених колекції можна зберігати об'єкти будь-якого типу. Для того щоб зберегти об'єкти класів, що визначаються користувачем, в типізованій колекції, доведеться скористатися класами узагальнених колекцій.

У колекції, для якої оголошено тип елементів, завдяки поліморфізму можна зберігати елементи будь-якого похідного класу, але не елементи інших типів.

Здавалося б, у порівнянні зі звичайними колекціями це обмеження, а не універсальність, однак на практиці колекції, в яких дійсно потрібно зберігати значення різних, не пов'язаних межу собою типів, майже не використовуються. Перевагою ж такого обмеження є те, що компілятор може виконати контроль типів під час компіляції, а не виконання програми, що підвищує її надійність і спрощує пошук помилок.

Використання стандартних узагальнених колекцій для зберігання і обробки даних є хорошим стилем програмування, оскільки дозволяє скоротити терміни розробки програм і підвищити їх надійність. Рекомендується ретельно вивчити властивості і методи цих класів і вибирати найбільш підходящі в залежності від розв'язуваної задачі.

Розглянемо детально клас **List<T>**. У класі **List<T>** реалізується узагальнений динамічний масив.

В класі **List<T>** визначено наступні властивості:

```
int Count {get; }
bool IsReadOnly {get; }
```

Властивість **Count** містить ряд елементів, що зберігаються в даний момент в колекції. А властивість **IsReadOnly** має логічне значення **true**, якщо колекція доступна тільки для читання. Якщо ж колекція доступна як для читання, так і для запису, то ця властивість має логічне значення **false**.

У класі **List<T>** визначається також властивість **Capacity**. Це властивість оголошується наступним чином.

```
public int Capacity {get; set; }
```

Властивість **Capacity** дозволяє встановити і отримати ємність колекції в якості динамічного масиву. Ця ємність дорівнює кількості елементів, які може містити колекція до її вимушеного розширення. Така колекція розширюється автоматично, і тому встановлювати її ємність вручну необов'язково. Але з міркувань ефективності це іноді можна зробити, якщо заздалегідь відомо кількість елементів колекції. Завдяки цьому виключаються витрати на виділення додаткової пам'яті.

У класу **List<T>** є такі конструктори.

```
public List ()
public List (IEnumerable <T> collection)
public List (int capacity)
```

Перший конструктор створює порожню колекцію класу клас List <T>з обраною за замовчуванням ємністю. Другий конструктор створює колекцію типу **List** з кількістю елементів, яка визначається розміром параметра **collection**. Третій конструктор створює колекцію типу **List**, що має початкову ємність, яка задається параметром **capacity**.

Ємність колекції, яка створюється у вигляді динамічного масиву, може збільшуватися автоматично при додаванні в неї елементів.

У класі **List<T>** визначається ряд методів, найбільш важливі з яких перераховані в табл. 5.2.

Таблиця 5.2 – Методи, визначені в класі **List <T>**

Метод	Опис
void Add (T item)	Додає елемент item в колекцію. Генерує виняток NotSupportedException, якщо колекція доступна тільки для читання
void Clear ()	Видаляє всі елементи з колекції
bool Contains (T item)	Повертає логічне значення true, якщо колекція містить елемент item, а інакше - логічне значення false
void CopyTo (T [] array, int arraylndex)	Копіює вміст колекції в масив array, починаючи з елемента, що вказується за індексом arraylndex
bool Remove (T item)	Видаляє перше входження елемента item в колекції. Повертає логічне значення true, якщо елемент item видалений. А якщо цей елемент не знайдено в колекції, то повертається логічне значення false
int IndexOf (T item)	Повертає індекс першого входження елемента item в колекції. Якщо елемент item не виявлений, то метод повертає значення -1
void Insert (int index, T item)	Вставляє в колекцію елемент item за індексом index
void RemoveAt (int index)	Видаляє з колекції елемент, розташований за індексом index
public virtual void AddRange (ICollection collection)	Додає елементи з колекції collection типу ArrayList
public virtual int BinarySearch (T item)	Виконує пошук в колекції значення, що задається параметром item. Повертає індекс знайденого елемента. Якщо шукане значення не знайдено, повертається від'ємне значення. Список повинен бути відсортований
public List <T> GetRange (int Index, int count)	Повертає частину колекції. Частина колекції починається з елемента за індексом index, і включає кількість елементів, що задається параметром count

public int IndexOf (T item)	Повертає індекс першого входження елемента item в колекції. Якщо шуканий елемент не виявлений, повертається значення -1
public void InsertRange (int index, IEnumerable <T> collection)	Вставляє елементи колекції collection починаючи з елемента за індексом index
public itemj int LastlndexOf (T	Повертає індекс останнього входження елемента item в колекції. Якщо шуканий елемент не виявлений, повертається значення -1
public void RemoveRange (int index, int count)	Видаляє частину колекції, починаючи з елемента за індексом index, і включаючи кількість елементів, яка визначається параметром count
public void Reverse ()	Повертає колекцію в зворотному порядку
public void Reverse (int index, int count)	Повертає колекцію в зворотному порядку, починаючи з елемента за індексом index, і включаючи кількість елементів, яке визначається параметром count
public void Sort ()	Сортує колекцію за зростанням
public void Sort (IComparer <T> comparer)	Сортує колекцію, використовуючи для порівняння спосіб, що задається параметром comparer. Якщо параметр comparer має пусте значення, то для порівняння використовується спосіб, який обирається за замовчуванням
public T [] ToArrayO	Повертає масив, який містить копії елементів колекції

Для циклічного звернення до елементів колекції, що представляє собою групу об'єктів, служить оператор **foreach**. Нижче наведена загальна форма оператора циклу **foreach**.

```
foreach (тип імя_змінної_циклу in ім’я_колекції)
оператор;
```

Тут тип *імя_змінної_циклу* позначає тип і ім'я змінної управління циклом, яка отримує значення наступного елемента колекції на кожному кроці виконання циклу **foreach**. А *ім’я_колекції* позначає колекцію, що циклічно переглядається. Отже, тип змінної циклу повинен відповідати типу елемента колекції. Крім того, тип може позначатися ключовим словом **var**. В цьому

випадку компілятор визначає тип змінної циклу, виходячи з типу елемента колекції.

Оператор циклу **foreach** діє таким чином. Коли цикл починається, перший елемент колекції вибирається і присвоюється змінній циклу. На кожному наступному кроці ітерації вибирається наступний елемент колекції, який зберігається в змінній циклу. Цикл завершується, коли всі елементи колекції виявляться переглянутими. Отже, оператор **foreach** циклічно опитує колекції по окремих її елементів від початку і до кінця.

Слід, однак, мати на увазі, що змінна циклу в операторі **foreach** служить тільки для читання. Це означає, що, присвоюючи цій змінній нове значення, не можна змінити вміст масиву.

Незважаючи на те що цикл **foreach** повторюється до тих пір, поки не будуть переглянуті всі елементи колекції, його можна завершити достроково, скориставшись оператором break.

Оператор **foreach** допускає циклічне звернення до масиву тільки в певному порядку: від початку і до кінця масиву, тому його застосування здається, на перший погляд, обмеженим. Але насправді це не так. У великій кількості алгоритмів, найпоширенішим з яких є алгоритм пошуку, потрібно саме такий механізм.

Розглянемо приклад програми, в якій показана робота з колекцією об’єктів класу *Lamp*.

```
using System;
using System.Collections.Generic;

namespace ClassElementTest
{
    class Lamp
    {
        private string _name;
        private double _power;

        public double Power
        {
            get { return _power; }
            set
            {
```

```
                if (value >= 0)
                {
                    _power = value;
                }
            }
        }

        public double Voltage { get; set; }

        public double Current
        {
            get { return _power / Voltage; }
        }

        public Lamp(string nameValue, double powerValue, double voltageValue)
        {
            _name = nameValue;
            Power = powerValue;
            Voltage = voltageValue;
        }

        public override string ToString()
        {
            return String.Format("Лампа {0}: потужність - {1} Вт, напруга -
{2} В, струм - {3:n3} А", _name, Power, Voltage, Current);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            //Створити список
            var lampList = new List<Lamp>();
            //Додати елементи в список
            lampList.Add(new Lamp("СЛ-21", 15, 220));
            lampList.Add(new Lamp("ПЛ-345", 75, 220));
            lampList.Add(new Lamp("РПС", 6, 12));
            lampList.Add(new Lamp("ДМ45", 45, 24));
            lampList.Add(new Lamp("Е100", 100, 220));
            //Вивести список на екран використовуючи цикл for
            Console.WriteLine("Список ламп за допомогою циклу for");
            for (int i = 0; i < lampList.Count; i++)
            {
                Console.WriteLine(lampList[i]);
            }
            //Вивести список на екран використовуючи цикл for
            Console.WriteLine("Список ламп за допомогою циклу foreach");
            foreach (var lamp in lampList)
            {
                Console.WriteLine(lamp);
            }
            Console.ReadLine();
        }
    }
}
```

У розглянутій програмі є все-таки один не зовсім очевидний недолік: створену колекцію *lampList* не можна відсортувати. Справа в тому, що в класах **ArrayList і List<T>** відсутні засоби для порівняння об'єктів користувацького типу даних. Для виходу з цього становища в класі можна реалізувати інтерфейс **IComparable**, в якому визначається метод порівняння об'єктів даного класу. Якщо потрібно впорядкувати об'єкти, що зберігаються в узагальненій колекції, то для цієї мети доведеться реалізувати узагальнений варіант інтерфейсу **IComparable<T>**. У цьому інтерфейсі визначається наведена нижче узагальнена форма методу **CompareTo()**:

```
int CompareTo(T other)
```

У методі CompareTo () поточний об'єкт порівнюється з іншим об'єктом other. Для сортування об'єктів по наростаючій конкретна реалізація даного методу повинна повертати нульове значення, якщо значення порівнюваних об'єктів рівні; додатнє – якщо значення поточного об'єкта більше, ніж у об'єкту other; і від'ємне – якщо значення поточного об'єкта менше, ніж у іншого об'єкта other.

Реалізуємо метод **CompareTo()** для класу *Lamp*, виконавши порівняння ламп за потужністю. Тобто будемо вважати, що більшою є лампа у якої більша потужність.

```
using System;
using System.Collections.Generic;

namespace ClassElementTest
{
    class Lamp:IComparable<Lamp>
    {
        private string _name;
        private double _power;

        public double Power
        {
            get { return _power; }
            set
            {
                if (value >= 0)
                {
                    _power = value;
                }
            }
```

```
        }

        public double Voltage { get; set; }

        public double Current
        {
            get { return _power / Voltage; }
        }

        public Lamp(string nameValue, double powerValue, double voltageValue)
        {
            _name = nameValue;
            Power = powerValue;
            Voltage = voltageValue;
        }

        public int CompareTo(Lamp other)
        {
            if (Power==other.Power)
            {
                return 0;
            }
            else if (Power>other.Power)
            {
                return 1;
            }
            else
            {
                return -1;
            }
        }

        public override string ToString()
        {
            return String.Format("Лампа {0}: потужність - {1} Вт, напруга -
{2} В, струм - {3:n3} А",
                _name, Power, Voltage, Current);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            //Створити список
            var lampList = new List<Lamp>();
            //Додати елементи в список
            lampList.Add(new Lamp("СЛ-21", 15, 220));
            lampList.Add(new Lamp("ПЛ-345", 75, 220));
            lampList.Add(new Lamp("РПС", 6, 12));
            lampList.Add(new Lamp("ДМ45", 45, 24));
            lampList.Add(new Lamp("Е100", 100, 220));
            //Список до сортування
            Console.WriteLine("Список ламп до сортування");
            foreach (var lamp in lampList)
            {
```

```
                Console.WriteLine(lamp);
            }
            //Відсортуємо список
            lampList.Sort();
            Console.WriteLine("Відсортований список ламп");
            foreach (var lamp in lampList)
            {
                Console.WriteLine(lamp);
            }
            Console.ReadLine();
        }
    }
}
```

Робоче завдання

Навчитися створювати колекції в мові програмування C# та отримати досвід використання колекцій для зберігання та роботи з сукупністю екземплярів класів.

Хід роботи

1. Модифікувати та доповнити програму створену під час виконання лабораторної роботи №4:
    - у методі *Main* () класу *Program* створити колекцію типу **List<T>** із екземплярами описаного класу;
    - додати в колекцію 5-10 нових елементів використовуючи методи **Add()** та **Insert()**;
    - вивести значення елементів колекції на екран за допомогою одного з циклів **for, while, do/while** на вибір;
    - вивести значення елементів колекції на екран за допомогою циклу **foreach**;
    - доповнити клас реалізацією інтерфейсу **IComparable<T>** шляхом створення методу **CompareTo()** з порівнянням по одній із властивостей на вибір;
    - у методі *Main* () класу *Program* виконати сортування створеної колекції та вивести результат на екран.

Індивідуальне завдання

Відповідно до завдання лабораторної роботи №1.

Контрольні питання

1. Які структури даних ви знаєте?
2. Які операції можна виконувати над списками?
3. Опишіть структуру даних стек.
4. Опишіть структуру даних черга.
5. Що таке колекція?
6. Які типи колекцій реалізовано в мові C#?
7. Що таке неузагальнені колекції?
8. Що таке узагальнені колекції?
9. Перерахуйте найважливіші колекції, описані в просторі System.Collections.Generic.
10. Які типи даних можна зберігати в узагальнених колекціях?
11. Які основні властивості визначені в класі **List<T>**?
12. Опишіть основні методи, визначені в класі **List<T>**.
13. Опишіть роботу оператора **foreach.**
14. Наведіть синтаксис оператора **foreach.**
15. Як забезпечити можливість сортування колекції з типом даних, що визначений користувачем?

ЛАБОРАТОРНА РОБОТА №6.
ФАЙЛОВЕ ВВЕДЕННЯ ТА ВИВЕДЕННЯ. СЕРІАЛІЗАЦІЯ ОБ'ЄКТІВ

Мета

Вивчити основні принципи роботи з файлами та серіалізації об'єктів в мові програмування C#. Отримати досвід збереження окремих екземплярів класу та колекцій і з допомогою бінарного форматера та у файлах XML.

Стислі теоретичні відомості

У середовищі .NET Framework передбачені класи для організації введення-виведення в файли. На рівні операційної системи файли складаються з байтів, тому існують відповідні методи для введення і виведення байтів в файли. Крім того, існують спеціальні операції символьного введення-виведення в файл, які застосовуються при обробці тексту.

Для створення байтового потоку, пов'язаного з файлом, служить клас **FileStream**. Клас **FileStream** визначено в просторі імен **System.IO**. Тому на початку будь-якої програми, що його використовує клас **FileStream**, додається такий рядок коду:

```
using System.IO;
```

Для відкриття файлу створюється об'єкт класу **FileStream**. В цьому класі визначено кілька конструкторів. Нижче наведено найпоширеніший серед них:

```
FileStream (string шлях, FileMode режим)
```

де *шлях* позначає ім'я файлу, включаючи повний шлях до нього; а *режим* порядок відкриття файлу, де вказується одне зі значень, що перерахуванні табл. 6.1. Як правило, цей конструктор відкриває файл для доступу з метою читання або запису. Винятком з цього правила є відкриття файлу в режимі **FileMode.Append**, коли файл стає доступним тільки для запису.

Таблиця 6.1 Значення режимів відкриття файлів FileMode

Значення	Опис
FileMode.Append	**Додає дані в кінець файлу**
FileMode.Create	**Створює новий вихідний файл. Існуючий файл з таким же ім'ям буде знищено**
FileMode.CreateNew	**Створює новий вихідний файл. Файл з таким же ім'ям не повинен існувати**
FileMode.Open	**Відкриває існуючий файл**
FileMode.OpenOrCreate	**Відкриває файл, якщо він існує. В іншому випадку створює новий файл**
FileMode.Truncate	**Відкриває існуючий файл, але скорочує його довжину до нуля**

Якщо спроба відкрити файл виявляється невдалою, то генерується помилка:

- якщо файл не можна відкрити тому що він не існує, генерується помилка **FileNotFoundException**;
- якщо файл не можна відкрити через помилки введення-виведення, генерується помилка **IOException**.

До числа інших помилок, які можуть бути згенеровані при відкритті файлу, відносяться такі: **ArgumentNullException** (вказано порожнє ім'я файлу), **ArgumentException** (вказано невірне ім'я файлу), **ArgumentOutOfRangeException** (вказана невірний режим), **SeaurityException** (у користувача немає прав доступу до файлу), **PathTooLongException** (занадто довге ім'я файлу або шлях до нього), **NotSupportedException** (в імені файлу вказано пристрій, який не підтримується), а також **DirectoryNotFoundException** (вказано невірний каталог).

По завершенні роботи з файлом його слід закрити, викликавши метод **Close()**.

У класі **FileStream** визначені два методу для читання байтів з файлу: **ReadByte()** і **Read()**. Так, для читання одного байта з файлу використовується метод **ReadByte()**, загальна форма якого наведена нижче:

```
int ReadByte()
```

Коли цей метод викликається, з файлу зчитується один байт, який потім повертається у вигляді цілого значення.

Для читання блоку байтів з файлу служить метод **Read()**, загальна форма якого виглядає так.

```
int Read(byte [] array, int offset, int count)
```

У методі **Read()** робиться спроба зчитати кількість *count* байтів в масив *array*, починаючи з елемента *array*[*offset*]. Метод повертає кількість байтів, успішно зчитаних з файлу. Якщо ж виникає помилка введення-виведення, то генерується виключення **IOException**.

Для запису байта в файл служить метод **WriteByte()**. Нижче наведена його найпростіша форма.

```
void WriteByte(byte value)
```

Цей метод виконує запис в файл байта, що позначається параметром *value*.

Для запису в файл цілого масиву байтів може бути викликаний метод **Write()**. Нижче наведена його загальна форма.

```
void Write(byte [] array, int offset, int count)
```

У методі **Write()** робиться спроба записати в файл кількість *count* байтів з масиву *array*, починаючи з елемента *array*[*offset*]. Метод повертає кількість байтів, успішно записаних в файл.

По завершенні виведення в файл, його слід закрити за допомогою методу **Close()**.

Незважаючи на те, що файли часто обробляються побайтово, для цього можна скористатися також символьними потоками. Перевага символьних потоків полягає в тому, що вони оперують символами безпосередньо в Unicode. Якщо потрібно зберегти текст, то для цього найкраще підійдуть саме символьні потоки. В цілому, для виконання операцій символьного введення-виведення в файли використовуються класи **StreamReader** та **StreamWriter**.

Для створення символьного потоку виведення досить укласти об'єкт класу **Stream**, наприклад **FileStream**, в оболонку класу **StreamWriter**. У класі **StreamWriter** визначено кілька конструкторів. Нижче наведено найпоширеніший серед них:

```
StreamWriter(Stream потік)
```

де *потік* позначає ім'я відкритого потоку. Цей конструктор генерує виняток **ArgumentException**, якщо потік не відкритий для виведення, а також виключення **ArgumentNullException**, якщо потік виявляється порожнім.

У деяких випадках зручніше відкривати файл засобами самого класу **StreamWriter**. Для цього служить один з наступних конструкторів:

```
StreamWriter(string шлях)
StreamWriter(string шлях, bool append)
```

де *шлях* – це ім'я файлу, включаючи повний шлях до нього. Якщо в другій формі цього конструктора значення параметра *append* рівне **true**, то дані приєднуються в кінець існуючого файлу. В іншому випадку ці дані перезаписують вміст зазначеного файлу. Але незалежно від форми конструктора файл створюється, якщо він не існує.

Нижче наведено простий приклад програми виведення на диск опису об’єкту класу *Lamp* у вигляді текстових рядків, як зберігаються в файлі test.txt. Для символьного виводу в файл в цій програмі використовується об'єкт класу FileStream, укладений в оболонку класу StreamWriter.

```
using System;
```

```
using System.IO;

namespace ClassElementTest
{
    class Lamp
    {  ...  }

    class Program
    {
        static void Main(string[] args)
        {
            var sampleLamp = new Lamp("СЛ-21", 15, 220);
            var fout=new StreamWriter("text.txt");
            fout.WriteLine(sampleLamp);
            fout.Close();
        }
    }
}
```

Для створення символьного потоку введення досить укласти байтовий потік в оболонку класу **StreamReader**. У класі **StreamReader** визначено кілька конструкторів. Найбільш часто використовується конструктор:

```
StreamReader(Stream потік)
```

де *потік* позначає ім'я відкритого потоку. Цей конструктор генерує виняток **ArgumentNullException**, якщо потік виявляється порожнім, а також виключення **ArgumentException**, якщо потік не відкритий для введення.

Іноді файл простіше відкрити, використовуючи безпосередньо клас **StreamReader**, аналогічно класу **StreamWriter**. Для цієї мети служить наступний конструктор:

```
StreamReader(string шлях)
```

де *шлях* – це ім'я файлу, включаючи повний шлях до нього. Вказаний файл повинен існувати. В іншому випадку генерується виняток **FileNotFoundException**. Якщо шлях виявляється порожнім, то генерується виключення **ArgumentNullException**.

Нижче наведено простий приклад програми зчитування з диску текстових рядків, як зберігаються в файлі *text.txt*.

```
using System;
```

```
using System.IO;
class Program
{
    static void Main()
    {
        var fin = new StreamReader(@"D:\text.txt");
        while (!fin.EndOfStream)
        {
            var s = fin.ReadLine();
            Console.WriteLine(s);
        }
        fin.Close();
    }
}
```

Зверніть увагу на те, як в цій програмі визначається кінець файлу. Доступна для читання властивість **EndOfStream** має логічне значення **true**, коли досягається кінець потоку, в іншому випадку – логічне значення **false**. Отже, властивість **EndOfStream** можна використовувати для відстеження кінця файлу.

У середовищі .NET Framework також визначено клас **File**, який є корисним для роботи з файлами, оскільки він містить статичні методи, що виконують типові операції над файлами. Зокрема, в класі **File** є методи для копіювання та переміщення, шифрування і розшифрування, видалення файлів, а також для отримання і завдання інформації про файлах, включаючи відомості про їхнє існування, часу створення, останнього доступу і різні атрибути файлів (тільки для читання, прихованих і ін.). Крім того, в класі **File** є ряд зручних методів для читання з файлів і записи в них, відкриття файлу і отримання посилання типу **FileStream** на нього.

Ряд методів для роботи з файлами визначено також в класі **FileInfo**. Цей клас відрізняється від класу **File** дуже важливою перевагою: для операцій над файлами він надає методи екземпляра і властивості, а не статичні методи. Тому для виконання декількох операцій над одним і тим же файлом краще скористатися класом **FileInfo**.

У C# є можливість зберігати не тільки дані примітивних типів, але і об'єкти. Термін серіалізація описує процес збереження (і, можливо, передачі) стану об'єкта в потоці (наприклад, файловому потоці і потоці в пам'яті). Послідовність даних, що зберігається містить всю інформацію, необхідну для реконструкції (або десеріалізації) стану об'єкта з метою подальшого використання. Застосовуючи цю технологію, дуже просто зберігати великі обсяги даних (в різних форматах) з мінімальними зусиллями. У багатьох випадках збереження даних програми з використанням серіалізації виливається в код меншого обсягу, ніж застосування класів для читання/запису з простору імен **System.IO**.

Щоб зробити об'єкт доступним для серіалізації, необхідно позначити клас (або структуру) атрибутом **[Serializable]**. Атрибути – це додаткові відомості про клас, які зберігаються в його метаданих. Якщо існують поля, які не повинні (або не можуть) брати участь в серіалізації, можна помітити такі поля атрибутом **[NonSerialized]**.

Об'єкти можна зберігати в одному з таких форматів: бінарному, SOAP або у вигляді XML-файла. Перераховані можливості представлені такими класами: **BinaryFormatter, SoapFormatter, XmlSerializer.**

Тип **BinaryFormatter** серіалізує стан об'єкта в потік, використовуючи компактний бінарний формат. Цей тип визначений в просторі імен **System.Runtime.Serialization.Formatters.Binary**. Таким чином, щоб отримати доступ до цього типу, необхідно вказати наступну директиву **using**:

```
using System.Runtime.Serialization.Formatters.Binary;
```

Тип SoapFormatter зберігає стан об'єкта у вигляді повідомлення SOAP. Цей тип визначений в просторі імен **System.Runtime.Serialization.Formatters.Soap**, що знаходиться в окремій збірці. Тому для форматування об'єктів в повідомлення SOAP необхідно спочатку встановити посилання на

System.Runtime.Serialization.Formatters.Soap.dll, використовуючи діалогове вікно *Add Reference* (Додати посилання) в Visual Studio і потім вказати наступну директиву **using**:

```
using System.Runtime.Serialization.Formatters.Soap;
```

Для збереження об'єктів в документі XML передбачений тип **XmlSerializer**. Щоб використовувати цей тип, потрібно вказати директиву **using**:

```
using System.Xml.Serialization;
```

Двома ключовими методами типу **BinaryFormatter** є: **Serialize()** – зберігає об'єкт в зазначений потік у вигляді послідовності байтів; **Deserialize()** – перетворює збережену послідовність байтів в об'єкт.

Припустимо, що після створення екземпляра класу і модифікації деяких даних стану потрібно зберегти цей екземпляр у файлі *.dat. Почати слід з створення самого файлу *.dat. Для цього можна створити екземпляр типу **System.IO.FileStream**. Потім потрібно буде створити екземпляр **BinaryFormatter** і передати йому **FileStream** і об'єкт для збереження.

Для прикладу розглянемо додаток в якому реалізована серіалізація об’єктів типу Lamp.

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

namespace ClassElementTest
{
    [Serializable]
    class Lamp
    {   ...   }
    class Program
    {
        static void Main(string[] args)
        {
            var sampleLamp = new Lamp("СЛ-21", 15, 220);
            var binFormat = new BinaryFormatter();
            var fStream = new FileStream("Lamp.dat", FileMode.Create);
```

```
            binFormat.Serialize(fStream, sampleLamp);
            fStream.Close();
        }
    }
}
```

Після виконання програми можна переглянути вміст файлу *Lamp.dat* в папці *bin\Debug* поточного проекту.

Тепер припустимо, що необхідно прочитати збережений об'єкт із двійкового файлу назад в змінну. Після відкриття файлу *Lamp.dat* (за допомогою методу **File.OpenRead()**) можна викликати метод **Deserialize()** класу **BinaryFormatter**. **Deserialize()** повертає об'єкт загального типу **System.Object**, тому необхідно застосувати явне приведення, як показано нижче:

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

namespace ClassElementTest
{
    [Serializable]
    class Lamp
    {  ...  }
    class Program
    {
        static void Main(string[] args)
        {
            var dataFile = File.OpenRead(@"D:\Lamp.dat");
            var binFormat = new BinaryFormatter();
            var sampleLamp = (Lamp)binFormat.Deserialize(dataFile);
            Console.WriteLine(sampleLamp);
        }
    }
}
```

Серіалізация об'єктів з використанням XmlSerializer може використовуватися для збереження стану заданого об'єкта у вигляді чистої XML-розмітки. XML-документ складається із тексту, і придатний до читання людиною. Розглянемо наступний код:

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
```

```
using System.Xml.Serialization;

namespace ClassElementTest
{
    [Serializable]
    public class Lamp
    {   ...   }
    class Program
    {
        static void Main(string[] args)
        {
            {
                var sampleLamp = new Lamp("СЛ-21", 15, 220);
                var xmlFormat = new XmlSerializer(typeof (Lamp));
                var xmlFile = new FileStream("Lamp.xml", FileMode.Create);
                xmlFormat.Serialize(xmlFile, sampleLamp);
                xmlFile.Close();
            }
            {
                var xmlFile = File.OpenRead(@"D:\Lamp.xml");
                var xmlFormat = new XmlSerializer(typeof(Lamp));
                var sampleLamp = (Lamp)xmlFormat.Deserialize(xmlFile);
                Console.WriteLine(sampleLamp);
            }
        }
    }
}
```

Ключова відмінність, від прикладу з **BinaryFormatter** полягає в тому, що тип **XmlSerializer** вимагає вказівки інформації про клас, який необхідно серіалізувати. У створеному файлі XML знаходяться показані нижче дані XML:

```
<?xml version="1.0"?>
<Lamp xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
  <Power>15</Power>
  <Voltage>220</Voltage>
</Lamp>
```

Клас **XmlSerializer** вимагає, щоб всі серіалізовані типи об'єктів підтримували стандартний конструктор (без параметрів).

Особливістю сереалізації колекцій об'єктів, є те, що більшість типів з просторів імен **System.Collections** і **System.Collections.Generic** вже позначені

атрибутом **[Serializable]**. Таким чином, щоб зберегти сукупність об'єктів, можна просто додати їх в колекцію (таку як звичайний масив, **ArrayList** або **List<T>**) і серіалізувати даний об'єкт в бажаний потік (файл).

Робоче завдання

Навчитися виконувати введення та виведення даних з файлів в мові програмування C# та отримати досвід використання серіалізації окремих об’єктів та колекцій.

Хід роботи

1. Модифікувати та доповнити програму створену під час виконання лабораторної роботи №5:
    - надати створеному класу специфікатор **public** та атрибут **[Serializable]**;
    - створити один екземпляр класу та вивести текстову інформацію про нього у файл. Відкрити файл у текстовому редакторі та перевірити наявність інформації. Навести вміст файлу у протоколі виконання роботи;
    - створити один екземпляр класу та виконати його серіалізацію у бінарному форматі у файл. Знайти файл на диску та перевірити наявність даних;
    - зчитати серіалізований об’єкт в бінарному форматі з файлу у нову змінну;
    - виконати серіалізацію раніше створеної колекції об’єктів у файл формату XML Відкрити файл у текстовому редакторі та перевірити наявність інформації. Навести вміст файлу у протоколі виконання роботи.

Індивідуальне завдання

Відповідно до завдання лабораторної роботи №1.

Контрольні питання

1. Що таке файл?
2. Який клас служить для байтового файлового введення/виведення?
3. Який клас служить для символьного файлового введення/виведення?
4. Як відкрити файл для зчитування?
5. Як відкрити файл для запису?
6. Які є режими відкриття файлу?
7. Які помилки можливі при роботі з файлами?
8. Як зчитати байти або символи з файлу?
9. Як записати байти або символи у файл?
10. Як визначається кінець файлу?
11. Для чого призначені класи **File** та **FileInfo**?
12. Що таке серіалізація?
13. Які види серіалізації доступні у мові C#?
14. Яка послідовність дій виконується для серіалізації об’єкта?
15. Яка послідовність дій виконується для десеріалізації об’єкта?
16. Як виконується серіалізація та десеріалізація колекцій?

ЛАБОРАТОРНА РОБОТА №7.
СТВОРЕННЯ ПРОЕКТУ WINDOWS FORMS. КЛАС APPLICATION

Мета

Познайомитися з основними можливостями розробки графічного інтерфейсу користувача з використанням технології Windows Forms. Отримати досвід створення простих віконних додатків. Вивчити базові елементи Windows Forms..

Загальні відомості про Windows Forms

Windows Forms дозволяє розробляти інтелектуальні клієнти. **Інтелектуальний клієнт** - це програма з багатим графічним інтерфейсом, проста в розгортанні і оновленні, здатна працювати при наявності або відсутності підключення до Інтернету, яка використовує більш безпечний доступ до ресурсів на локальному комп'ютері в порівнянні з традиційними додатками Windows.

Windows Forms є технологією інтелектуальних клієнтів для .NET Framework; це набір керованих бібліотек для спрощення реалізації програмних задач, наприклад читання і запис в файлову систему. За допомогою середовища розробки типу Visual Studio можна створювати додатки Windows Forms, які відображають інформацію, запитують введення користувачів і обмінюються даними з віддаленими комп'ютерами по мережі.

У Windows Forms форма є видимою поверхнею, на якій відображається інформація для користувача. Зазвичай додаток Windows Forms будується шляхом розміщення елементів управління на формі і написанням коду для реагування на дії користувача, такі як клацання миші або натискання клавіш. Елемент управління - це окремий елемент інтерфейсу, призначений для відображення або введення даних.

При виконанні користувачем якої-небудь дії з формою або одним з її елементів управління, створюється подія. Додаток реагує на ці події за допомогою коду і обробляє події при їх виникненні. Додаткові відомості див Створення обробників подій в Windows Forms.

Windows Forms включає широкий набір елементів управління, які можна додавати на форми: текстові поля, кнопки, списки, що розкриваються, перемикачі та навіть веб-сторінки. Список всіх елементів управління, які можна використовувати в формі, см Елементи управління для використання в формах Windows Forms. Якщо існуючий елемент управління не задовольняє потребам, в Windows Forms можна створити власні настроюються елементи управління за допомогою класу UserControl.

До складу Windows Forms входять елементи призначеного для користувача інтерфейсу з розширеними функціями, відповідними можливостям потужних додатків, таких як Microsoft Office. Використовуючи елементи управління ToolStrip і MenuStrip, можна створювати панелі інструментів і меню, що містять текст і малюнки, що відображають підміню і містять в собі інші елементи управління, такі як текстові поля і поля з списком, що випадає.

За допомогою конструктора Windows Forms Visual Studio, що підтримує перетягування, можна легко створювати додатки Windows Forms: Досить виділити елемент керування курсором і помістити його на потрібне місце на формі. Конструктор надає такі кошти, як лінії сітки і "прив'язка ліній" для подолання труднощів вирівнювання елементів управління. І в разі використання Visual Studio або компіляції з командного рядка можна використовувати елементи управління FlowLayoutPanel, TableLayoutPanel і SplitContainer для створення просунутих розміток форми за мінімальний час і з мінімальними зусиллями.

System.Windows.Forms - простір імен

Простір імен System.Windows.Forms містить класи для створення додатків Windows, які дозволяють Найбільш ефективно використовувати Розширені можливості призначеного для користувача інтерфейсу, які надаються операційною системою Microsoft Windows.

У наступній таблиці наведено згруповані за категоріями класи, що відносяться до простору імен System.Windows.Forms.

Категорія класів	Докладні відомості
Елементи управління, призначені для користувача елементи управління і форми	Більшість класів в просторі імен **System.Windows.Forms** є похідними від класу Control .Клас Control надає основні функціональні можливості для всіх елементів управління, які відображаються в Form .
Меню і панелі інструментів	Windows Forms включає широкий набір класів, які дозволяють створювати призначені для користувача панелі інструментів і меню, що відрізняються сучасним вигляд і поведінкою. ToolStrip , MenuStrip , ContextMenuStrip і StatusStrip дозволяють створювати панелі інструментів, терміни меню, контекстні меню і терміни стану, відповідно.
Елементи управління	Простір імен **System.Windows.Forms** надає велику кількість класів елементів управління, які дозволяють створювати призначені для користувача інтерфейси з розширеним можливостями.
Макет	Кілька принципових класів в Windows Forms допомагають контролювати розташування елементів управління на Отображаемое поверхні наприклад в формі або елементі управління. На панелі FlowLayoutPanel послідовно розміщені всі елементи управління, які вона містить.

Дані і прив'язка даних	Windows Forms забезпечує розширену архітектуру для прив'язування до таких джерел даних, як бази даних і XML-файли. Елемент управління DataGridView надає настроюється таблицю для відображення даних і дозволяє налаштовувати формат осередків, термін, стовпців і кордонів.
Компоненти	Крім елементів управління простір імен **System.Windows.Forms** надає інші класи, які НЕ є похідними від класу Control , але також забезпечують візуальні функції для додатків Windows.Такі класи, як ToolTip і ErrorProvider , розширюють можливості або Предоставляютя відомості користувачам.

У просторі імен **System.Windows.Forms** є ряд класів, що забезпечують підтримку вищезазначених класів.Прикладами підтримує класів можуть служить перерахування, класи аргументів подій і Делегати, які використовуються подіями в елементах управління і компонентах.

Створення простої форми в Windows Forms

Створення форми Windows Forms

1. Запустіть Visual Studio.
2. Створіть додаток Windows з ім'ям HelloWorld.
3. З **панелі елементів** перетягнути в форму елемент управління **Button.**
4. Виділити його клацанням миші.У вікні "Властивості" надайте властивості Text значення "Say Hello".

Написання коду програми

1. Двічі Клацніть кнопку, щоб додати оброблювач подій Click .Відкриється редактор коду, при цьому положення курсора виявиться всередині обробника подій.

2. Вставте цей код:

MessageBox.Show ("Hello, World!");

Тестування додатка

1. Натисніть клавішу F5, щоб запустить додаток.
2. Коли додаток запущено, натисніть кнопку і перевірте, чи відображається фраза "Hello, World!"
3. Закрийте форму Windows Forms, щоб повернутися в Visual Studio.

Створення обробника подій за допомогою конструктора

1. Клацніть форму або елемент керування, для якого потрібно створити обробник подій.
2. У вікні Властивості натисніть кнопку Події ().
3. У списку доступних подій Клацніть подія, для якого потрібно створити обробник подій.
4. В поле праворуч від імені події введіть ім'я обробника і натисніть клавішу ENTER.

З'явиться Редактор коду з кодом для форми; метод обробника подій створюється в коді за аналогією з кодом в наступному прикладі:

```
private void StartProcess (object sender, System.EventArgs e) {
   // Add event handler code here.
}
```

5. Додайте Відповідний код в обробник подій.

Порядок виконання роботи

1. Ознайомитися з теоретичним матеріалом.
2. Самостійно вивчити і описати властивості і методи класу Application.
3. За допомогою вікна Properties досліджувати властивості і події форми.

4. Вибрати індивідуальне завдання, номер завдання відповідає номеру студента в групі.
5. Налагодити програму на комп'ютері.
6. Підготувати звіт по роботі.

Індивідуальні завдання:

1. При переміщенні миші по формі виводити її координати в підпис форми.(MouseMove, MouseEventArgs)
2. При натисканні кнопки змусить рухатися форму по колу.
3. При натисканні правої кнопки миші збільшити ширину форми на 10 лівій зменшити.
4. При натисканні правої кнопки миші збільшити висоту форми на 10 лівій зменшити.
5. При натисканні кнопки миші форма повинна згортатися (Minimized).
6. При натисканні кнопки миші форма повинна змінювати стиль кордону (FormBorderStyle).
7. При натисканні правої кнопки миші форма повинна переміщатися вправо, лівої - вліво.
8. При натисканні правої кнопки миші форма повинна переміщатися вгору, лівою - вниз.
9. При натисканні кнопки миші випадковим чином змінювати колір форми.
10. При натиснутій правій кнопці миші форма повинна переміщатися слідом за нею.
11. При натисканні кнопки миші випадковим чином змінювати положення форми.
12. При натисканні кнопки миші переміщати форму по квадрату.
13. При натисканні правої, лівої кнопки миші змінювати колір форми.

14. При натисканні правої кнопки миші форма повинна переміщатися вправо, лівої - вліво.
15. При натисканні правої кнопки миші форма повинна переміщатися по діагоналі.
16. При натисканні кнопки миші поміняти колір форми.
17. При натисканні кнопки миші форма повинна збільшувати площу в 2 рази.
18. При натисканні кнопки миші випадковим чином змінювати розміри форми.
19. При натисканні правої кнопки миші форма повинна переміщатися вправо, лівої - вліво.

ЛАБОРАТОРНА РОБОТА №8.
ВИДИ ЕЛЕМЕНТІВ УПРАВЛІННЯ І РОБОТА З НИМИ

Мета

Познайомитися з розширеними можливостями розробки графічного інтерфейсу користувача з використанням технології Windows.Forms. Навчитися використовувати різні елементи управління. Отримати досвід використання діалогових вікон для відкриття та збереження файлів. Навчитися створювати меню програми.

Теоретичні відомості

В ході розробки і зміни призначеного для користувача інтерфейсу додатків Windows Forms потрібно додавати, вирівнювати і розміщувати елементи управління. Елементи управління - це об'єкти, які знаходяться всередині об'єктів форми. Кожен тип елемента управління має власний набір властивостей, методів і подій, що відповідають певному призначенню. З елементами управління можна працювати в конструкторі або додавати їх динамічно під час виконання з допомогою коду.

Функціональна класифікація елементів управління Windows Forms

У Windows Forms існують елементи управління і компоненти, що виконують ряд функцій. У наступній таблиці представлений список компонентів і елементів управління Windows Forms відповідно до основною функцією. Крім того, якщо для виконання однієї і тієї ж функції служать кілька елементів управління, то рекомендовані елементи управління перераховані із зазначенням використовувалися раніше застарілих елементів управління. Застарілі елементи управління також перераховані в окремій таблиці; поруч з кожним таїмо елементом управління вказано новий елемент, що прийшов на зміну застаріле.

Функція	Елемент управління	Опис
Відображення даних	DataGridView	Елемент управління DataGridView надає настроюється таблицю для відображення даних.Клас DataGridView забезпечує настройку осередків, термін, стовпців і меж таблиці. **Примітка** елемент управління DataGridView підтримує ряд простих і складних функцій, відсутніх в елементі управління DataGrid .Додаткові відомості див. В розділі Відмінності елементів управління DataGridView і DataGrid в Windows Forms .
Прив'язка даних і переміщення	BindingSource	Спрощує прив'язку елементів управління у формі до даних завдяки засобам управління грошовими одиницями, повідомлення про зміни і т.д.
	BindingNavigator	Надає інтерфейс, подібний панелі інструментів, для переходів по формі і управління даними.
редагування тексту	TextBox	Відображає текст, введений в режимі розробки, Який користувачі можуть змінювати під час виконання або за допомогою програмних засобів.
	RichTextBox	Дозволяє представлять текст в текстових форматі або в форматі RTF.
	MaskedTextBox	Обмежує формат даних, що вводяться користувачем.
Відображення інформації	Label	Відображає текст, недоступний для безпосереднього редагування користувачем.

(тільки для читання)		
	LinkLabel	Відображає текст у вигляді веб-посилання і створює подія при клацанням тексту. Як правило, текст є посиланням на інше вікно або на веб-вузол.
…	…	…

Додавання елементів керування в форми Windows Forms.

1. Відкрийте форму.
2. В **панелі елементів** Клацніть елемент керування, Який потрібно додати в форму.
3. Клацніть місце в формі, де повинен розташовуватися лівий верхній кут елемента керування, а потім перетягнути покажчик миші на місце, в якому повинен розташовуватися правий нижній кут елемента керування.

Елемент управління додається на форму у зазначеній місце з Зазначеними розмірами.

Порядок виконання роботи

1. Ознайомитися з теоретичним матеріалом.
2. Створити проект WindowsApplication.
3. Вивчити властивості та події класу запропонованого об'єкта управління
4. Написати програму на мові C #.
5. Підготувати звіт по роботі.

Індивідуальні завдання.

1. Елемент управління Button.Створити форму з кнопкою. При натисканні на будь-яку кнопку форми створювати нову кнопку **(SuspendLayout)**

2. Елемент управління CheckedListBox.Створити форму з CheckedListBox з декількох пунктів. При зміні кількості зазначених пунктів, виводити це кількість в підпис форми.
3. Елемент управління CheckBox.Створити форму з CheckBox при зміні властивості Checked міняти колір форми.
4. Компонент ColorDialog.Створити форму з кнопкою. При натисканні кнопки викликати ColorDialog, і поміняти колір форми відповідно до обраним.
5. Елемент управління ComboBox.Створити форму з ComboBox і TextBox, при зміні значення TextBox і натисканні клавіші Enter додавати це значення в ComboBox.
6. Елемент управління ContextMenuStrip.Створити контекстне меню для управління раз мерами форми.
7. Елемент управління DateTimePicker.Створити форму з двома DateTimePicker. При зміні дати в одному з них у другому відображати дату найближчого вихідного.
8. Елемент управління Label.Створити форму з Кількома Label. При виборі мишкою будь-якого з них відображати його текст в заголовку форми.
9. Елемент управління ListBox.Створити форму з ListBox і TextBox, при зміні значення TextBox і натисканні клавіші Enter додавати це значення в ListBox.
10. Елемент управління PictureBox.Використовуючи PictureBox і OpenFileDialog створити переглядач зображень.
11. Елемент управління MonthCalendar.Створити форму з MonthCalendar. При зміні значення MonthCalendar виводити це значення і поточний час в заголовок форми.
12. Компонент Timer.Використовуючи Timer виводити поточний час в заголовок форми.
13. Елемент управління MenuStrip.Створити меню для управління кольором форми
14. Елемент управління RadioButton.Помістити три RadioButton на один GroupBox, при виборі однієї з RadioButton - виводити її текст в заголовок форми.
15. Елемент управління ProgressBar.При кожному натисканні по формі мишкою змусить рухатися повзунок ProgressBar. При досягненні повзунком крайнього положення обнуляти значення.
16. Елемент управління RichTextBox.Відображати кожну нову інформацію, що вводиться в RichTextBox букву новим кольором з періодом 5.

17. Елемент управління TextBox.За допомогою FontDialog і кнопки змінювати фонт для елемента управління TextBox.
18. Елемент управління TabControl.Змінюючи закладки TabControl - міняти стиль кордону форми.
19. Елемент управління TreeView.Скласти програму заповнення дерева за допомогою TextBox і Button. Причому, новий текст додавати в якості підлеглого в виділений вузол, якщо виділений є (св-во **SelectedNode),** і в якості основного якщо виділеного коментарі.
20. Елемент управління CheckBox.При зміні властивості CheckBox - міняти значення ширини і висоти форми один з одним.
21. Компонент FontDialog.За допомогою FontDialog міняти фонт підпису кнопки на формі.
22. Елемент управління ComboBox.За допомогою ComboBox міняти підпис форми.
23. Елемент управління RadioButton.Міняти колір форми за допомогою GroupBox і кілька RadioButton.
24. Елемент управління TextBox.Створити форму з двома TextBox. При зміні значення в одному з них, виводити теж значення, але в зворотному порядку в іншому.

ЛАБОРАТОРНА РОБОТА №9.
ЗНАЙОМСТВО З ІНТЕРФЕЙСОМ GDI +

Мета

Познайомитися з інтерфейсом графічних пристроїв GDI +. Навчитися будувати графічні примітиви на формах Windows.Forms.

Теоретичні відомості

Простір імен System.Drawing забезпечує доступ до функціональних можливостей графічного інтерфейсу GDI +. Простору імен System.Drawing.Drawing2D, System.Drawing.Imaging, і System.Drawing.Text забезпечують додаткові функціональні можливості.

Клас Graphics надає методи малювання на пристрої відображення. Такі класи, як Rectangle і Point інкапсулюють елементи GDI +. Клас Pen використовується для малювання ліній і кривих, а класи, похідні від абстрактного класу Brush, використовуються для заливки фігур.

Клас	Опис
Bitmap	Інкапсулює точковий малюнок GDI +, що складається з даних точок графічного зображення і атрибутів малюнка. об'єкт Bitmap використовується для роботи з зображеннями, які визначаються даними точок.
Brush	Визначає об'єкти, які використовуються для заливки всередині графічних фігур, таких як прямокутники, еліпси, кола, багатокутника і доріжки.
Brushes	Кисті для кожного зі стандартних кольорів. Цей клас бути не може бути успадкований.
BufferedGraphics	Надає графічний буфер для подвійний буферизації.
BufferedGraphicsContext	Надає методи створення графічних буферів, які можуть вживатися для подвійний буферизації.
BufferedGraphicsManager	Надає доступ до об'єкта основного контексту буферизує графіки для домену додатки.

ColorConverter	Перетворює кольори одного типу даних в інший. Доступ до даного класу здійснюється за допомогою об'єкта TypeDescriptor .
ColorTranslator	Здійснює переклад кольорів в структури GDI + Color і з них.Цей клас бути не може бути успадкований.
Font	Визначає конкретний формат тексту, включаючи Нарис шрифту, його розмір і атрибути стилю. Цей клас бути не може бути успадкований.
FontConverter	перетворює об'єкти Font з одного типу даних в інший.
FontConverter. FontNameConverter	Інфраструктура. FontConverter.FontNameConverter - перетворювач типу, Який використовується для перетворення імені шрифту в інші різні уявлення і назад.
FontConverter. FontUnitConverter	Інфраструктура. Перетворює одиниці шрифту в інші типи одиниць і назад.
FontFamily	Визначає групу гарнітур зі схожим базовим конструктором і певними відмінностями в стилі. Цей клас бути не може бути успадкований.
Graphics	Інкапсулює поверхню малювання GDI +. Цей клас не можна успадкувати.
…	…

Структура	Опис
CharacterRange	Визначає діапазон позицій символу в межах рядка.
Color	Являє кольору в термінах каналів альфа, червоного, зеленого і синього (ARGB).
Point	Являє впорядковану пару цілих чисел - координат X і Y, визначальну точку на двовимірної площині.
PointF	Являє впорядковану пару координат X і Y з плаваючою комою, що визначає точку на двовимірної площині.
Rectangle	Містить набір з чотирьох цілих чисел, що визначають розташування і розмір прямокутника. Для розширення функцій області Використовуйте об'єкт Region .
RectangleF	Містить набір з чотирьох цифр з плаваючою комою, що визначають розташування і розмір прямокутника. Для розширення функцій області Використовуйте об'єкт Region .

Size	Зберігає впорядковану пару цілих чисел, зазвичай ширину і висоту прямокутника.
SizeF	Містить впорядковану пару чисел з плаваючою комою, зазвичай ширину і висоту прямокутника.

Приклад який малює зелений круг на формі

```
// Подія форми Paint
 private void Form1_Paint (object sender, PaintEventArgs e) {
    // Create pen.
    Pen blackGreen = new Pen (Color.Green, 3);

    // Create rectangle for ellipse.
    Rectangle rect = new Rectangle (50, 50, 100, 100);

    // Draw ellipse to screen.
   e.Graphics.DrawEllipse (blackGreen, rect);

 }
```

Порядок виконання роботи

1. Ознайомитися з теоретичним матеріалом.
2. Вибрати варіант завдання по номеру в групі.
3. Написати програму на мові C #.
4. Налагодити програму на комп'ютері.
5. Підготувати звіт по роботі.

Індивідуальні завдання

1. Намалювати "цукерку": прямокутник з проведеними діагоналями і два рівнобедрених трикутника, що примикають до нього з боків.Лівий нижній кут прямокутника має координати (X, Y), довжина сторони A. Висоти трикутників A / 2. Зафарбувати утворені діагоналями і сторонами прямокутника два протилежних трикутника.
2. Завдання про мильні бульбашки.Спочатку на екрані з'являється в довільному (випадковому) місці точка, вона потім розростається в коло (видувається мильна бульбашка) і зникає (лопається), видаючи звук. Коло

лопається, якщо його радіус стає більше R_0 або воно стосується стінок (кордонів екрану).Цей цикл повторюється, поки не буде натиснута будь-яка клавіша.

3. Написати програму, яка виводила б в графічному режимі на весь екран монітора, крім прямокутника (100,100) - (300,200), випадково точки до тих пір, поки Ви не натиснете Esc.

4. Намалювати вежу з трьома зубцями і аркою.Лівий нижній кут вежі має координати (X, Y), довжина основи A. Інші розміри досить довільні, але повинні бути виражені через A. Вежу зафарбувати.

5. Намалювати куб.Передня ліва нижня вершина куба має координати (X, Y), довжина ребра A. зафарбувати верхню межу, а в правій грані провести діагоналі.

6. Намалювати пряму шестикутну піраміду.Передній лівий кут підстави піраміди має координати (X, Y), довжина ребра підстави A. Інші розміри досить довільні, але повинні бути виражені через A. Дві бічні грані піраміди зафарбувати.

7. Намалювати пряму усічену чотирикутну піраміду.Передній лівий кут підстави піраміди має координати (X, Y), довжина ребра підстави A, верхній грані - A / 2.Інші розміри досить довільні, але повинні бути виражені через A. Праву бічну грань піраміди зафарбувати.

8. Написати програму, яка малювала забарвлену "товсту" букву "E".Ординати і абсциси точок, що описують букву, повинні бути описані як масив.

9. Написати програму, яка малювала 7 зафарбованих ялинок.Ялинки повинні бути подібні один одному, розташовані уздовж горизонталі на одній висоті, висота ялинок повинна лінійно збільшуватися зліва направо.

10. Написати програму, яка виводила б в графічному режимі на весь екран монітора, крім кола з центром (200,200) і радіусом 80, випадково точки до тих пір, поки Ви не натиснете Esc.

11. Написати програму, яка виводила б на весь екран монітора в графічному режимі випадково зірочки, поки не буде натиснута будь-яка клавіша.

12. Намалювати (досить простий) автомобіль.Автомобіль повинен проїхати по екрану справа наліво.

13. Написати програму, яка малювала забарвлену ялинку.

14. Написати програму, при натисканні клавіші "p" малювала б коло, при натисканні клавіші "b" малювала б прямокутник, при натисканні клавіші "f" малювала б зафарбований прямокутник, при натисканні клавіші Esc завершувала б роботу.Після малювання кожної фігури вона повинна стиратися.

15. Написати програму, яка виводила б в графічному режимі на весь екран монітора випадково точки до тих пір, поки Ви не натиснете Esc.

16. Намалювати шахову дошку.

17. Намалювати гору цегли.На вершині гори лежить одну цеглину, під нею - дві, під двома - три, під трьома - чотири і т.д. Висота гори - 10 цеглин. Гора має бути симетричною.

18. Використовуючи тільки один оператор циклу намалювати сходи з 10 сходинок, парні і непарні ступені якої мають різні розміри.

19. Використовуючи один оператор циклу намалювати "спіраль Архімеда»: один крок вгору, два кроки вліво, три кроки вниз, чотири кроки вправо, п'ять кроків вгору тощо.

20. Написати програму, яка виводила б в графічному режимі на екран монітора випадково п'ятикутні зірки до тих пір поки не буде натиснута клавіша Esc.

21. Написати програму, яка виводить на екран зображення пучка з 20 випадкових прямих, що виходять з однієї випадкової точки, поки Ви не натиснете ESC.

22. Намалювати п'ятикутну зірку, вписану в коло.

23. Намалювати послідовність веж.Кожна вежа являє собою поставлені один на одного квадрати (поверху). Кожен наступний квадрат менше попереднього в 2 рази. Вежа симетрична.

24. Намалювати пряму усічену чотирикутну призму.Передній лівий кут підстави призми має координати (X, Y), довжина ребра підстави A, верхній межі - A / 2. Інші розміри досить довільні, але повинні бути виражені через A. Ліву бокову грань призми зафарбувати.

25. Написати програму, яка б малювала "товсту" букву M, рух якої по екрану управляється клавішами ArrawLeft, ArrawRight, ArrawUp, ArrawDown.

ЛАБОРАТОРНА РОБОТА №10.
ПОБУДОВА ГРАФІКІВ ФУНКЦІЙ

Мета

Познайомитися з підходами до побудови графіків. Навчитися будувати графіки функцій з використанням мови C# та технології Windows.Forms.

Теоретичні відомості

При вирішенні наукових, навчальних та ділових програм часто потрібно побудувати графіки різних функцій. функції `y = f` (x), де «`x`» - незалежна змінна (аргумент) і «`y`» - залежна змінна (метод) можуть бути задані в програмі по-різному. Це може бути якась формула (аналітичний метод), наприклад, `y = sin` (x). Інший спосіб полягає в тому, що змінні «x», «y» можуть бути задані таблично, іншими словами, у вигляді масивів `x (i)` і `y` (i), де `i` - 0, 1, 2, ..., `Npoints` (число точок графіка в цьому випадку дорівнює `Npoints` + 1, а число відрізків між точками графіка одно Npoints).

У даній роботі пропонується побудувати графік для функцій, заданих таблично.

Порядок виконання

1. Використовуючи варіант індивідуального завдання розробити програму, яка відображає графік y=f(x) з використанням Windows.Forms
2. Забезпечити можливість зміни аргументу x в форммі з автоматичним оновленням графіку.

Для виконання даної роботи необхідно буде познайомитися з елементом `PictureBox,` Який в C # використовується для відображення графіки, з його властивостями і методами.

Оскільки потрібно побудувати графік для функції, заданої у вигляді таблиці, потрібно буде також вивчити способи роботи з таблицями в C # - класом `DataGridView,` з його властивостями і методами.

Робота з таблицями даних

Нижче наводиться ряд рекомендацій для організації введення даних. Для роботи з ним необхідно створити довільний текстовий файл, який містить табличну інформацію функції (значення аргументу і значення функції) і помістити його в папку роботи. Дані у файлі повинні представляти собою набір агрументів і значеннь в рядку Повинні розділятися одним пропуском.

1. Створіть форму, на яку помістити наступні компоненти:

- таблицю `DataGridView`
- `TextBox` в режимі `multiLine`
- 2 кнопки з іменами:

  `btnFileToGrid` - для читання даних з файлу в таблицю,

  `btnGridToTextBox` - для читання даних з таблиці.

- керуючий елемент `OpenFileDialog` для зручності вибору потрібного файлу з набором даних.

2. Таблиця `DataGridView` має властивості `Columns` і `Rows`, що представляють набори її стовпців і рядків, які нумеруються з нулю. Відкривши в вікні `Properties` для властивості `Columns` редактор (при натисканні на знак крапки), створіть два стовпці для значень аргументу і значення функції відповідно. Введіть для їх властивостей `HeaderText` і `Name` наступні значення:

	0-й стовпець	1-й стовпець
HeaderText	аргумент	функція
Name	ColumnArg	ColumnFun

Зрозуміло, при виконанні роботи слід створити таблицю з 4 стовпцями і з відповідною назвами.

3. Не забудьте, що файл з даними бажано розмістити в той же папці, що і проект. Натиснувши на кнопці btnFileToGrid, отримаєте шаблон обробника події для читання даних з файлу в таблицю. Напишіть цей обробник, Використовуючи наступний алгоритм:

- Перевірити, правильність відкриття файлу, обраного в діалогів вікні:

  ```
  if (openFileDialog1.ShowDialog () == DialogResult.OK)
  ```

- Створити екземпляр класу `StreamReader.`
- У циклі (Ще не дійшли до кінця файлу) вважати послідовно всі терміни з файлу, виконавши для них такі перетворення.
- Кожну зчитану терміну перетворити в масив підрядків за допомогою методу `Split` для змінних типу String. Ці подстроки і будуть представлять собою значення в осередках (шпальтах) для кожного рядка в таблиці. У нашому прикладі стовпців в кожному рядку - 2, а для файлу роботи - чотири.
- Щоб записати ці значення в таблицю, потрібно скористатися класом `C #`, Який представляє собою колекцію термін для класу DataGridView.Це клас DataGridViewRowCollection, Який має метод, що дозволяє додавати терміни до колекції, Яким ми і Скористаємося.

Нижче показано створення екземпляра цього класу (з ім'ям `rows)`, які пов'язують з термінами таблиці `dataGridView` за допомогою покажчика this.

```
DataGridViewRowCollection rows = this.dataGridView.Rows;
```

Написане вище пропозиція заснована на тому, що в `C #` всі екземпляри (об'єкти) класів є ссилочними типами.Тому воно означає, що, виконуючи дії над об'єктом `rows`, ми звертаємося за посиланням до нашої таблиці - її властивості Rows.Це `забезпечує неявна змінна this`, що

представляє собою покажчик на той екземпляр класу, Який був використаний при виклику методу.

Метод, що дозволяє додавати терміни до колекції, Яким можна скористатися для запису перетворених термін з файлу в таблицю, має такий вигляд: `rows.Add(row)`

Після виконання цих дій таблиця виявиться **заповненою** даними з файлу.

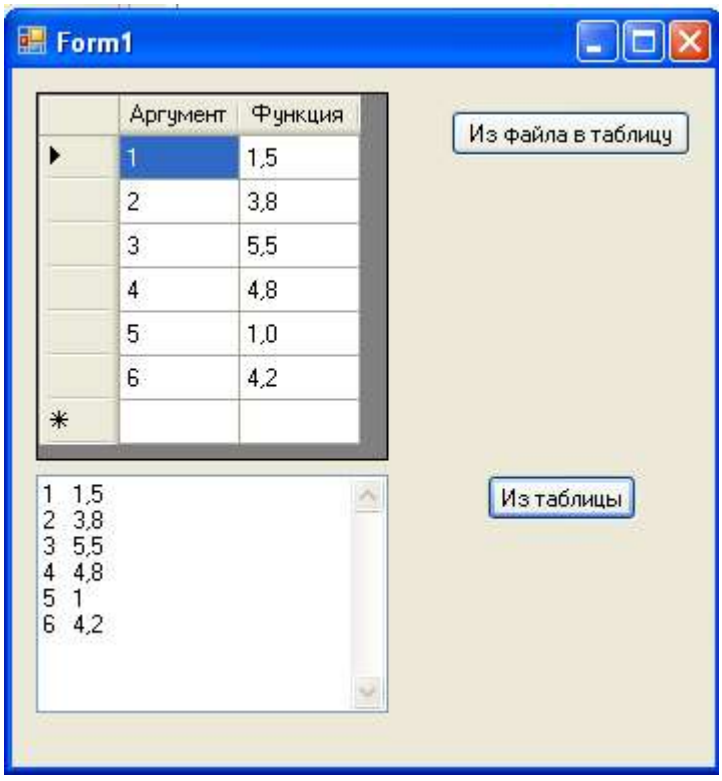
Тепер потрібно навчитися **вибирати** дані з таблиці. У запропонованій задачі дані Повинні бути записані в поле TextBox. Для виконання цих дій призначений другий обробник події - натискання на кнопку btnGridToTextBox.

Що потрібно пам'ятати?

Таблиця - це набір рядків. Як вже було сказано вище, вони нумеруються з 0. З іншого боку, таблиця - це набір осередків. Для кожного рядка кожна клітинка (cell) задається номером стовпця, які теж нумеруються з 0. Наприклад, для і-го рядка таблиці `DataGridView` У нашій прикладу звернення до нуля і на одну і-го рядка буде виглядати так:

```
double arg =Convert.ToDouble (DataGridView.Rows [i].Cells [0].Value); double fun = Convert.ToDouble (DataGridView.Rows [i].Cells [1].Value);
```

У нашій задачі (побудова графіка залежності) дані, що зберігаються в осередках таблиці, для виведення в поле `TextBox` повинні бути інтерпретовані в строковий тип. На малюнку наведено приклад ефектів у програмному забезпеченні для прикладу.



Робота з графікою

Наступним етапом роботи є побудова графіка функції, для значень, заданих в таблиці. Для цього на форму потрібно помістити Control-елемент `PictureBox`. Для того щоб цей елемент візуально виділявся на області форми, рекомендується спочатку на форму помістити елемент-контейнер `Panel`, у якого задати властивість `BorderStyle` рівним `FixedSingle`, а вже потім помістити на нього - `PictureBox`. В класі `PictureBox` визначено важлива властивість - `Graphics`. За допомогою численних параметрів класу `Graphics` можна виводити в область, зайняту елементом управління, текстові написи, геометричні фігури і різні зображення.

Клас `Graphics` передається в якості параметра `PaintEventArgs e` для події Paint, яке відіграє дуже важливу роль в додатках `Windows` Forms. Воно забезпечує **перемальовування** вікна програми, як тільки в цьому виникне необхідність, наприклад, якщо поле одного вікна було тимчасово зайняте іншим вікном, або потрібно висновок другого графіка.

Тому завдання побудови графіка рекомендується виконувати, викликаючи метод Paint.

Нижче наведено фрагмент коду для виведення зображення системи координат з впровадженням методу DrawLine (малювати лінію) класу Graphics.

Спочатку оголошується клас Graphics для елемента PictureBox1 з ім'ям grpBox, задається колір пера boxPen – чорний і товщина пера – 2 і потім малюються дві перпендикулярні лінії. Лінії малюються з впровадженням примірників класу Point (крапка).Точка задається двома координатами за допомогою оператора new. У визначенні координат використовуються властивості Width (ширина) і Height (висота) елемента PictureBox1.

```
private void pictureBox1_Paint (object sender, PaintEventArgs e)
{
     Graphics grpBox = e.Graphics;
     Pen boxPen = new Pen (Color.Black, 2) grpBox.DrawLine
     (boxPen,new Point (10 10), new Point (10 pictureBox1.Height -
     10));     grpBox.DrawLine     (boxPen,new     Point     (10,
     pictureBox1.Height - 10),new Point (pictureBox1.Width - 10
     pictureBox1.Height - 10))
}
```

Розглянемо докладніше питання про те, як були обрані координати точок - початку і кінця прямих для координатних осей.

Побудова графіків на екрані монітора має наступні важливі особливості.

Перша особливість полягає в тому, що інтерфейс графічного пристрою виводить графік (за замовчуванням) в системі координат, початок якої розташовано у верхньому лівому кутку області малювання .Ось x спрямована в цьому випадку вправо, а ось y - вниз. Коли, наприклад, ми будемо розміщувати графік в елементі управління pictureBox, то

початок координат буде розміщено саме в лівому верхньому кутку цього елемента. Для нашої задачі (число ітерацій і значення точності - позитивні величини) початок системи координат потрібно розмістити в лівому нижньому кутку. Тому початок системи координат буде в точці:

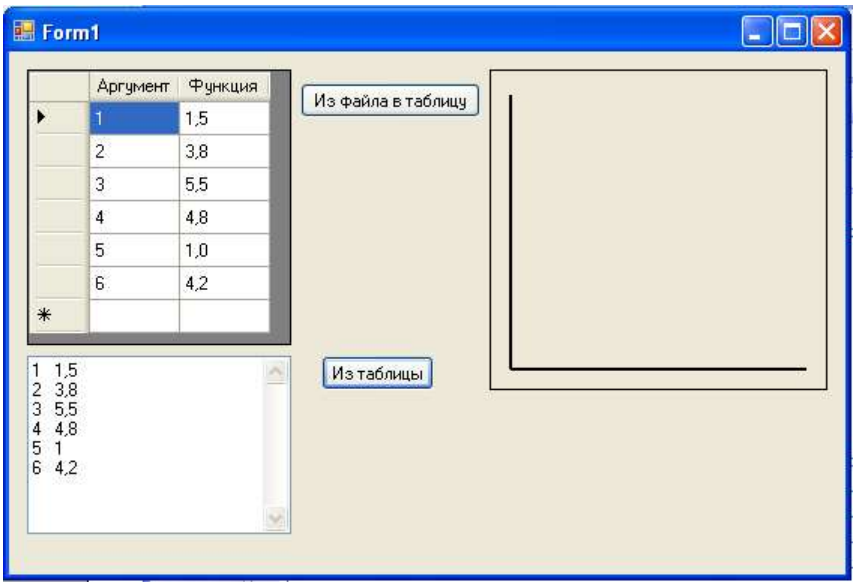
`point(10,pictureBox1.Height - 10).`

Константа `10` (пікселів) – це відступ від лівої і нижньої сторони `pictureBox` для розміщення написів поділів по осях. Кінцеві точки також вибираються з урахуванням цього відступу.

У загальному випадку за допомогою паралельного перенесення осей на величини `ox_pix` і `oy_pix` необхідно ввести нову систему координат, в якій ось «y» до того ж повинна бути спрямована вгору.

Друга особливість полягає в тому, що графік будується не в дійсних значеннях для функції y = f (x), які розрахує програми, а в пікселях. Тому Ми повинні підібрати масштаби `Mx` і `My` для перекладу дійсних значень змінних в пікселі з метою збільшення (або зменшення) розмірів графіка таким чином, щоб його було видно на екрані. Для змінних коду в пікселях введемо далі розширення «Pix».

На малюнку - результат роботи програми з впровадженням такого коду.



Правила побудови графіків

Щоб за пропонованою методикою можна було будувати графіки будь-яких функцій, замість дійсних значень пропонується використовувати відносні ***безрозмірні*** значення змінних.

1. Графік функції y = f (x) будується в відносних безрозмірних величинах (позначені зі штрихом) x` , y` :
x`= X / | x_max |; y` = Y / | y_max |,

де | x_max | і | y_max | - Найбільші абсолютні значення змінних x і y. Отже, будь-який графік буде знаходиться всередині прямокутника

1 = < x` <= 1;

1 = < y` <= 1

і стосуватися хоча б в одній точці двох горизонтальних прямих y`= 1 і y`= -1 і двох вертикальних x`= 1 і x`= 1.

2. Щоб Отримати дійсний значення x і y в будь-якій точці графіка, необхідно визначити відносні значення x` і y` в цій точці і помножити їх на Найбільше абсолютне значення | x_max | або | y_max |, відповідно.

3. Для отримання **координатної сітки,** в якій буде перебувати графік, кожну з двох осей координат x і y можна розбити на деяке число рівних частин, наприклад, на 20 рівних частин, і з точок розбиття провести горизонтальні і вертикальні лінії.

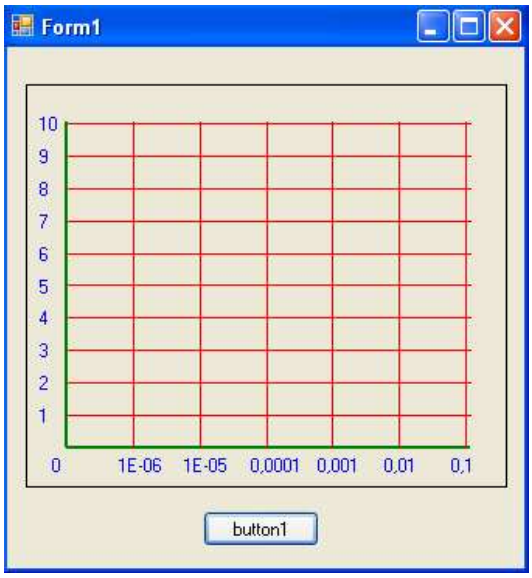
4. Поза координатної сітки слід записати цифрові або літерні позначення величин графіка.

Нижче наведено фрагмент програмного коду з коментарями, в якому побудована сітка для графіка функції, табличного значення якої отримані при обчисленнях суми нескінченного ряду (варіанта завдання).

```
public int NPoints = 6;

int indent_pix = 20; // Відступ потрібен для приміщення написів по осях public
float x_max = 0.1f; // значення x_max і x_min задані в умові завдання public float
x_min = 0.00001f;
int N_step_grid_x = 7, // Щоб було 6 вертик. ліній в сітці по умові
float step_grid_x;
public int step_grid_x_pix; // крок зміни значень аргументу в пікселах
public int step_grid_y_pix; // крок зміни значень функції в пікселах
int M_x = 250;
int M_y = 210;
int x_point_end_pix;
int y_max = 5, // Значення функції y_max і y_min.Для функції ліга y_max = 5
int y_min = 1, // y_max і y_min визначаються завданням до роботі


// Оброблювач події Paint для pictureBox1
private void pictureBox1_Paint (object sender, PaintEventArgs e)
{
// Створюємо свій екземпляр класу Graphics
Graphics myGraphics = e.Graphics;
// Можна створити екземпляр Graphics інакше (див. Наступний оператор)
// Graphics myGraphics = pictureBox1.CreateGraphics ();


// визначаємо положення осей координат в пікселях
int ox_pix = indent_pix; // координата x початку координат
int oy_pix = pictureBox1.Height - indent_pix; // коорд. y початку
координат float x_point_end; // Оголошення кінцевої точки по осі x
x_point_end = 1.0f; // Значення кінцевої точки по осі x з умові


// Обчислення значення кінцевої точки осі x в пікселах с впровадженням
// масштабного коефіцієнта і з явним перетворенням типу float в тип int


x_point_end_pix = (int) x_point_end * M_x - indent_pix;


// Вибираємо зелене перо товщиною 2;
Pen greenPen_x = new Pen (Color.Green, 2);
// Задаємо координати двох граничних точок осі:
Point point1 = new Point (ox_pix, oy_pix)
Point point2 = new Point (x_point_end_pix, oy_pix)
// Будуємо лінію через дві заданий граничні точки:
myGraphics.DrawLine (greenPen_x, point1, point2)


// Для нашої задачі число горизонтальних ліній в сітці одно y_max
int N_step_grid_y = y_max;
// Крок сітки в напрямку осі "y" (висота всієї сітки дорівнює 1 одиниці)
float step_grid_y; // крок зміни значень функції
```

```
int step_grid_y_pix; // крок зміни значень функції в пікселах step_grid_y
= 1.0f / N_step_grid_y; // значення кроку по осі y step_grid_y_pix =
(Int) (step_grid_y * M_y) // кроку по осі y в пікселах
// Вибираємо червоне перо товщиною 1:
Pen redPen = new Pen (Color.Red, 1);


// Будуємо в циклі горизонтальні лінії сітки від нульової лінії (осі x) вгору:
// Для цього визначаємо координати граничних точок сітки
int j_y; // лічильник для циклу
int y1_pix = oy_pix;
for (j_y = 1; j_y <= N_step_grid_y; j_y ++)
{
y1_pix = y1_pix - (int) step_grid_y_pix;
// Задаємо координати двох граничних точок лінії сітки: Point
point3 = new Point (ox_pix, y1_pix) // ліва крапка
Point point4 = new Point (x_point_end_pix, y1_pix) // права крапка
// Будуємо пряму лінію через дві заданий точки:
myGraphics.DrawLine (redPen, point3, point4)
}
// Будуємо ось ординат "oy" від y = 0 до y = 1;
// оголошую і задаємо ординату останньої точки осі ординат "y" при y = 1:
float y_point_end_pix; y_point_end_pix
= oy_pix;
// Вибираємо зелене перо товщиною 2;
Pen greenPen = new Pen (Color.Green, 2);
// Задаємо координати двох граничних точок осі y:
Point point5 = new Point (ox_pix, indent_pix) // Верхн. точка в pictureBox
Point point6 = new Point (ox_pix, (int) y_point_end_pix) // нижня
// Будуємо лінію через дві заданий граничні точки:
myGraphics.DrawLine (greenPen, point5, point6)


// Будуємо вертикальні лінії сітки від осі y вправо
int j_x; float x1; int
x1_pix;
step_grid_x = (Float) (1.0f / N_step_grid_x) // крок зрад. знач. аргумен.
step_grid_x_pix = (int) (step_grid_x * M_x) // крок зрад. в пікселах x1_pix
= ox_pix;
for (j_x = 1; j_x <N_step_grid_x; j_x ++)
{
x1_pix = x1_pix + (int) step_grid_x_pix;
// Задаємо координати двох граничних точок лінії сітки:
Point point7 = new Point (x1_pix, indent_pix) // ліва крапка
Point point8 = new Point (x1_pix, (int) y_point_end_pix) // права
// Будуємо пряму лінію через дві заданий точки:
myGraphics.DrawLine (redPen, point7, point8)
}


// записуємо числа по осях координат, користуючись функцією DrawString (msg, ...):
// оголошую локальні змінні:
```

```
int n; float p1 = 1.0f; float p2; string msg;


// записуємо числа по осі "+ oy":
for (n = 0; n <= N_step_grid_y - 1, n ++) //  9
{
// p2 = p1 - n * 0.1F; ця формула виводить знач. у вигляді беск. дробу, тому
// множимо обчислене значення на 100.0f, округляємо його за допомогою
// функції Math.Round і ділимо результат на 100.0f
// p2 = (float) (Math.Round ((p1 - n * (float) 1 / N_step_grid_y) * 10.0F)) / 10.0F;
p2 = (Float) (Math.Round
        ((p1 - n * (Float) 1 / N_step_grid_y) * 100.0F)) / 100.0F; msg =
Math.Round  (p2 * N_step_grid_y).ToString  (); // Отримуємо значення
myGraphics.DrawString (msg, this.Font, Brushes.Blue,
ox_pix - 20 oy_pix - 215 + N * step_grid_y_pix)
}
// записуємо числа по осі "+ ox":
f loat kf = 0.00001f; // Коеф. для виведення в зручному вигляді
p2 = 0.0f; // Нач. значення для виведення нуля на початку координат
for (n = 1; n <= 7; n ++)
{
msg = p2.ToString ();
myGraphics.DrawString  (msg, this.Font, Brushes.Blue,
ox_pix - 55 + n * step_grid_x_pix, oy_pix + 5);
// Щоб без нескінченної дробу вивелися все значення по осі "ox",
// Помножити і ділити доводиться на 1000000.0f
p2 = (Float) (Math.Round ((0.1F * kf) * 1000000.0f)) / 1000000.0f; kf
= kf * 10.0f;
} // ---------------------- Тепер будуємо графік ----------------
------- ----------
```



Індивідуальне завдання

№ вар.	Функція	x	y	Крок
1	$y = \cos 2x$	0..10	автомасштаб	0.1
2	$y = x^3 + 1$	-10..10	автомасштаб	0.2
3	$y = \sin 4x$	3..16	автомасштаб	0.3
4	$y = \sqrt{\cos x}$	-10..0	автомасштаб	0.1
5	$y = \ln(x + 1)$	0..10,	автомасштаб	0.2
6	$y = \lvert\cos x + \sin x\rvert$	-10..10	автомасштаб	0.3
7	$y = \sin 3x + x$	3..16	автомасштаб	0.1
8	$y = \cos \ln x$	0..6	автомасштаб	0.2
9	$y = \cos^2 x$	0..10,	автомасштаб	0.3
10	$y = x^2 + \lvert\cos x\rvert$	-10..10	автомасштаб	0.1
11	$y = \sin 5x$	0..10,	автомасштаб	0.2
12	$y = \text{arctg}\, \lvert x + 1\rvert$	-10..10	автомасштаб	0.3
13	$y = \cos x^3$	3..16	автомасштаб	0.1
14	$y = \sin e^x$	-10..0	автомасштаб	0.2
15	$y = \text{arctg}\, x + x$	0..10,	автомасштаб	0.3
16	$y = \sqrt{x} + \ln x$	-10..10	автомасштаб	0.1
17	$y = \cos 2x - \sin x$	3..16	автомасштаб	0.2
18	$y = \ln x + e^{\lvert x\rvert}$	0..6	автомасштаб	0.3
19	$y = \sin^2 x + \cos^2 x$	0..10,	автомасштаб	0.1
20	$y = -x^3 + 3x^2 + 3$	-10..10	автомасштаб	0.2
21	$y = \cos 2x - \sin^3 x$	3..16	автомасштаб	0.3
22	$y = \lvert\text{arctg}\, x\rvert$	-10..0	автомасштаб	0.1
23	$y = \cos^2 e^x$	0..10,	автомасштаб	0.2
24	$y = \lvert\cos x + 1\rvert$	-10..10	автомасштаб	0.3
25	$y = \sin 2x$	3..16	автомасштаб	0.1

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. Голуб Б.М. C#. Концепція та синтаксис. Навч. посібник. – Львів: Видавничий центр ЛНУ імені Івана Франка, 2006. – 136 с.
2. Троелсен Э. Язык программирования C# 5.0 и платформа .NET 4.5, 6-е изд.: Пер. с англ. – М.: ООО “И.Д. Вильямс”, 2013. – 1312 с.
3. Шилдт Г. C# 3.0. Полное руководство / Пер. с англ. – М.: Диалектика-Вильямс, 2009. – 992 с.
4. Котов, О.М. Язык C#: краткое описание и введение в технологии программирования: учебное пособие / О.М. Котов. – Екатеринбург: Изд-во Урал. ун-та, 2014. – 208 с.
5. Уотсон К. Microsoft Visual C# 2008. Базовый курс / К. Уотсон, К. Нейгел, Я.Х. Педерсен, Дж. Д. Рид, М. Скиннер, Э. Уайт. / Пер. с англ. – М.: Диалектика-Вильямс, 2009. – 1216 с.

Додаткова література:

1. Павловская Т.А. C#. Программирование на языке высокого уровня: Учебник для вузов. – СПб.: Питер, 2014. – 432 с.
2. Нейгел К., Ивьен Б., Глинн Д. C# 4.0 и платформа .NET 4 для профессионалов / Пер. с англ. – К.: Диалектика, 2011. – 1440 с.
3. Рихтер Дж. Программирование на платформе Microsoft .NET Framework 2.0 на языке C#. / Пер. с англ. – СПб: Питер, М: Русская Редакция, 2007. – 656 с.
4. Агапов В.П. Основы программирования на языке C#: учебное пособие / В. П. Агапов. – Москва: МГСУ, 2012. – 128 с.
5. Петцольдт Ч. Программирование для Microsoft Windows на C#. В 2-х томах. Том 1. /Пер. с англ. – М.: Издательско-торговый дом «Русская Редакция», 2002. – 576 с.
6. Андрианова А.А., Исмагилов Л.Н., Мухтарова Т.М. Объектно-ориентированное программирование на C#: Учебное пособие / А.А. Андрианова, Л.Н. Исмагилов, Т.М. Мухтарова. – Казань: Казанский (Приволжский) федеральный университет, 2012. – 134 с.
7. Вирт, Н. Алгоритмы и структуры данных [Текст]: пер. с англ. / Никлаус Вирт. – СПб: Невский Диалект, 2008. – 352 с.
8. Культин Н. Б. C# в задачах и примерах. – СПб.: БХВ-Петербург, 2007. – 240 с.
9. Ватсон К. C# / К. Ватсон, М. Беллиназо, О. Корне, Д. Эспиноза, 3. Гринфосс и др. / Пер. с англ. яз. – М.: Изд. «Лори». – 2005, 862 с.
10. Бокс Д., Селлз К. Основы платформы .NET, том 1. Общеязыковая исполняющая среда.: Пер. с англ. – М.: Издательский дом «Вильямс», 2003. – 288 с.
11. Либерти Дж., Программирование на C#. Создание .NET-приложений. Изд. 2-е. / Пер. с англ. – Спб.: Изд. «Символ», 2003. – 688 с.